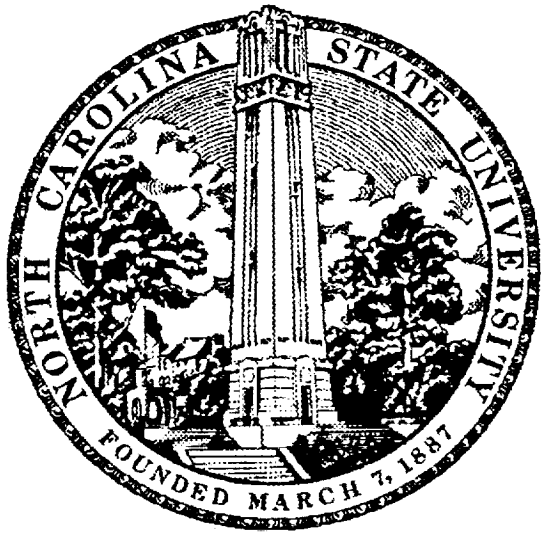
Computer Science
Technical Report



Experiments in Fault Tolerant Software Reliability

Report 5 -- NAG-1-667

April 1, 1989

David F .McAllister - Mladen A. Vouk

North Carolina State University

Box 8206
Raleigh, NC 27695

Annual Technical Report Submitted to the

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
Langley Research Center, Hampton, Va.

for research entitled

EXPERIMENTS IN FAULT TOLERANT SOFTWARE RELIABILITY

(NAG-1-667)

from

David F. McAllister, Co-Principal Investigator, Professor
Mladen A. Vouk, Co-Principal Investigator, Assistant Professor

Department of Computer Science
North Carolina State University
Raleigh, N.C. 27695-8206
(919) 737-2858

Report Period
Beginning Date: April 1, 1988.
Ending Date: March 31, 1989.

Raleigh, April 1, 1989

ANALYSIS OF FAULTS DETECTED IN A LARGE-SCALE MULTI-VERSION SOFTWARE EXPERIMENT*

Mladen A. Vouk, David F. McAllister
Amit M. Paradkar, Satyanarayana R. Vemulakonda

North Carolina State University,
Department of Computer Science, Box 8206
Raleigh, NC 27695-8206

Key Words: correlated faults, similar errors, software testing, multiversion testing, back-to-back testing, software fault-tolerance, software reliability

Abstract

Twenty functionally equivalent programs were were built and tested in a multiversion software experiment. Following unit testing, all programs were subjected to an extensive system test. In the process sixty-one distinct faults were identified among the versions. Less than 12% of the faults exhibited varying degrees of positive correlation. The common-cause (or similar) faults spanned as many as 14 components. However, majority of these faults were trivial, and easily detected by proper unit and/or system testing. Only two, of the seven similar faults, were "difficult" faults, and both were caused by specification ambiguities. One of these faults exhibited variable identical-and-wrong response span, i.e. response span which varied with the testing conditions and input data. Techniques that could have been used to avoid the faults are discussed. For example, it was determined that back-to-back testing of 2-tuples could have been used to eliminate about 90% of the faults. In addition, four of the seven similar faults could have been detected by using back-to-back testing of 5-tuples. We believe that most, if not all, similar faults could have been avoided had the specifications been written using more formal notation, the unit testing phase was subject to more stringent standards and controls, and better tools for measuring the quality and adequacy of the test data (e.g. coverage) were used.

*This research was supported in part by NASA grants NAG-1-667.

1. Introduction

The most common fault-tolerant software mechanisms are based on redundancy. They include N-version programming [Avi77, Avi84], recovery-block [Ran75], and some hybrid techniques [e.g. Sco87]. For acceptable results all of these techniques require the component failures to be mutually independent, or at least that the positive inter-component failure correlation is low [Eck85]. Some earlier experiments [Sco84, Vou85, Bis86,88, Kni86, Tso87, Avi88, Kel88, ShL88] have shown that failure dependence among functionally equivalent software components may not be negligible in the context of the current software development and testing techniques. This lead some authors to question the practical validity of the redundancy based fault-tolerance approaches [Kni86, ShL88].

In the period 1985-87 NASA LaRC funded a four-university[1] experiment to develop and validate functionally equivalent software versions. The goal of the experiment was to produce a set of software versions which could be used to study different aspects of multiversion software development and usage. In this paper we examine the character of the faults found in the twenty functionally equivalent versions generated during the experiment. The aim is to provide more information on which design and testing strategies might have been most suitable for avoiding these faults in the final product.

In the following section we describe the experiment in which the software components were produced. In section 3 we describe the profile of the detected faults, and we discuss the question of what could have been done to avoid these faults. The summary is in given in section 4. In this report we use the terms 'component' and 'version' as synonyms for the term 'functionally equivalent program, or code'.

2. Experiment

The programmers worked in two-person teams formed by random selection. All programmers worked from the same specification document written in English prose [CRA85]. A formal specification language was not used. The programming teams were responsible for the software design, the implementation and the unit testing phases of the life-cycle. The experimenters provided a

[1]Participating universities: North Carolina State University (NCSU), University of Illinois Urbana Champagne (UIUC), University of California Los Angeles (UCLA), University of Virginia (UVA). Participants from industry were: Research Triangle Institute (RTI), Charles River Analytics (CRA). The 1985 phase of the experiment was coordinated by RTI with assistance from CRA. The 1987 phase of the experiment was coordinated by CRA.

requirement specification document and acceptance (systems) testing of the product. The programs solved a problem in inertial navigation. The requirement was to interpret and analyze part of sensor signals (accelerometers only) received from a redundant strapped down inertial measurement unit (RSDIMU). It was also required that the code be written in Pascal, and developed and tested in a UNIX environment on VAX hardware. The problem specification was new, written for the experiment, and was not debugged via a "pilot" version of the code, or prototyping, prior to the production of the redundant versions. The initial (very limited) acceptance testing included two critical variable arrays, one 3 and the other 8 elements long, using 50 random test cases. The tolerance for comparison of floating point numbers was much larger than the accuracy of the input data. Functional and structural test coverage was low. We shall call this acceptance testing 'phase I acceptance testing'. Software development and testing was done on VAX 11/750 and 780 hardware running UNIX 4.2 BSD. The estimated reliability of the components based on **this** (very limited) acceptance testing was about 0.94 (see Appendix I for the estimation process). Additional information about the experiment can be found in [Kel86, Tso87, Kel88].

The phase II acceptance testing of the produced programs took place in the summer 1987. We shall call this phase 'certification' phase. The aim of this testing phase was not to evaluate any particular testing methodology, but to discover, and correct, any faults that may have been left in the versions after completion of the phase I testing so that we could study the nature of these faults. The base specification document [UCSB/CRA87] included the changes and additions to the specifications established in clarifications and ammendments sent to RSDIMU programmers during the phas I testing. The programs were assigned (see Appendix II) among twenty certification programmers located at NCSU, UCSB[1] and UVA. Each version was repeatedly tested against the "gold" code answers. The "gold" code, the testing harness, and the acceptance test data set were produced at NCSU and distributed to all sites. Whenever any differences were discovered, with respect to the golden answer, the programmers were requested to investigate the problem and correct the cause. They were given 20 consecutive difference at a time. Regression testing with the full acceptance test set was applied to all re-submitted software. Careful records were kept of any changes in the code, and the programmers were asked to submit a software change and correction report detailing the fault description, its symptoms and the changes. Intermediate code versions for seven NCSU programs (iterations) were also retained for further analysis.

In this phase we used tolerances compatible with the accuracy of the input data. All 11 output variables (59 individual values) were checked for each input. A difference was signaled whenever

[1]University of California Santa Barbara

any one of these values differed from the corresponding golden value. The results were obtained using a tolerance of 0.000244 relative (for 12 bits of input data accuracy) for real numbers larger than 0.1 in magnitude, and 0.0000244 absolute otherwise (requirements specification assumed that the cockpit display size was five digits), and 0 for integers. The test data sets consisted of 801 extremal and special value (ESV) and 400 random test cases which provided as complete functional and linear-block coverage as we believed was feasible under input variable value ranges dictated by the requirements specifications. The testing harness and the test data are described in Appendix III. Functional coverage was determined by comparison of the ESV cases and specifications, while cumulative execution coverage of the code and procedures was determined through execution tracing and post processing of the code and trace files. Unexecuted code was inspected and the reasons for its non-execution were determined after the certification phase ended. In several instances we found code portions dealing with functions not required by the specified problem (and therefore not tested for by ESV acceptance cases).

Table 2.1 Program characteristics[1].

Actual Name	Program	Total (loc)	Code (loc)	Percent comments	Exec. code	Linear blocks	Xqt (msec)	Percent Change
ncsua	P1	4165	3514	55.0	2052	513	496	+1.06
ncsub	P2	1809	1312	25.5	1132	315	880	+0.11
ncsuc	P3	4177	3376	47.6	3390	544	701	+3.45
ncsud	P4	1828	1570	39.5	1310	343	582	+7.21
ncsue	P5	2394	2218	39.7	1689	475	828	+3.63
uclaa	P6	2100	1781	37.3	1317	338	718	+7.36
uclab	P7	1833	1635	25.9	1492	368	510	+8.65
uclac	P8	1924	1563	35.7	1336	378	386	+2.66
uclad	P9	2525	2208	47.6	1481	432	394	-2.73
uclae	P10	1704	1351	33.9	1170	288	395	+1.85
uiuca	P11	4886	4209	85.4	3884	263	320	+1.03
uiucb	P12	2159	1558	46.9	1221	260	469	-37.63
uiucc	P13	1974	1609	55.3	1024	261	355	-1.20
uiucd	P14	2746	2389	51.3	1712	313	347	+17.55
uiuce	P15	2139	1787	31.2	1456	351	1283	+4.03
uvaa	P16	3208	2634	21.8	2379	469	4511	+2.82
uvab	P17	3198	2237	49.8	2141	436	694	+2.69
uvac	P18	3034	2723	27.4	2180	918	390	+8.66
uvad	P19	1837	1525	12.8	1389	319	1929	+0.98
uvae	P20	2686	2295	18.2	2317	501	521	+1.89

[1] Data reflects information for the certified versions as received from CRA.

Certification testing detected a number of faults of varying prevalence and seriousness. Some of the

faults were found to be highly correlated. One fault was detected in the testing harness and one fault was found (both by UCSB programmers) in the "golden" program through code and output inspections. In both cases the site responsible for the testing harness (NCSU) was notified immediately, the faults were corrected, and appropriate patches with an additional 5 test cases were distributed to all certification sites before the certification testing was concluded. These additional test cases were aimed at exercising the functionality that was missed in the golden code. The rationale for adding more ESV test cases to the 796 already in the ESV set was that the fault was discovered during the acceptance testing and therefore it should be removed from all versions, including the golden one, during that process. Unfortunately, examination of the versions received form CRA revealed that UCSB did implement these corrections, but UVA did not. This has to be taken into account when the "certified" versions are being analyxes (i.e. they do not have the same "base line"). The certification testing was done on VAX 11/785 hardware running UNIX 4.3BSD, and MicroVAX II hardware running Ultrix 1.2. The estimated reliability of the components based on phase II acceptance testing was ≥ 0.992 (see Appendix I).

2.1 Testing

The principal aim of the testing process reported here was to evaluate and classify all the faults that have been discovered during the certification phase. The results described in this paper are based on 801 special cases (796+5 ESV cases), 400 random cases used during certification testing, and two additional sets of 1000 random test cases each generated according to two random profiles. One generation profile, which we call RANDOM-Ib, was uniform over each individual variable including noise. The other, which we call RANDOM-II, had normally distributed calibration noise, exponentially distributed number of noisy sensors during calibration, and it was constructed so that the edge vector test never failed more than one additional sensor, and no sensors were failed on input. The random data were used to estimate software reliability and fault visibility, and served as a check on the extremal and special value testing.

The correctness of the answers was adjudicated using a "golden" program. This program has been very extensively tested and inspected on its own, and we believe that right now no faults remain in that code. However, as has been the case in all the experiments published so far [e.g. Pan81, Bis86,88, Avi88, ShL88] we have not proved the "golden" code correct.

The test sets we used here, and during phase II certification effort, were not designed to uncover any faults that may be associated with a series of time-correlated consecutive calls to the RSDIMU

algoritms (e.g. errors due to slowly accumulating deviations in variables, we have deliberately intitialized all variables on evry entry to RSDIMU procedure). Furthermore, although some faults related to numerical instabilities and round-off errors were uncovered in the algorithms the test sets were not designed to specifically search for such faults. We belive that we have detected all the faults that could be detected without using a complex flight simulator, but it is quite possible that some more subtle faults, including numerical round-off errors, could manifesting near, or below, the tolerance thresholds we used. In addition, time-correlation and subtle numerical faults may also appear if explicitly searched for, although for the latter the magnitudes of the differences between the corret and incorrect values are probably close the tolerances we used.

For each test case we generated a 21x20 response matrix. Each element of this matrix contained a vector of 59 values describing the relationship between the response of the row version and the column version. The vector contained one entry for each variable (or its part, if it was a compound variable) with 0 denoting agreement, and 1 disagreement between the answers. Row zero carried the responses of the versions to comparisons with the "golden" answers. We then combined the analysis of the matrices with the inspections of the actual response values, and the inspections of the code.

A brief structural profile of the versions investigated in the present experiment is given in Table 2.1. The column marked 'Total' gives the total number of lines of code (loc), including comments and blank lines. The column 'Code' gives Pascal code size excluding comments (obtained using pxp -s). The 'Percent comments' column indicates the comment level of the code[1]. The 'Linear blocks' column gives the count of linear code blocks as reported by the pxp processor, and the 'Xqt' column serves to illustrate the diversity of the average execution time per test case among the components. Note that one of the components is exceedingly inefficient, but since execution time constraints were not part of the specifications this is not considered an error. The 'Percent Change' column reflects the difference in the final code size (in number of lines, including comments) with respect to the initial version (sign indicating either additions or deletions).

Figure 2.1 illustrates the quality of the phase II acceptance test cases. We show the typical linear block coverage provided by the different components of the data set used. The coverage is expressed as the percentage of the covered linear blocks, where a linear block is as defined by the UNIX 'pxp' processor. We plot the linear block coverage (LBC) function

[1] It is the ratio of non-empty comment lines (including in-line comments) to the total number of non-empty lines of code (including comments)

$$LBC = \frac{\text{Number of executed linear-blocks}}{\text{Total number of linear-blocks}} \times 100.$$

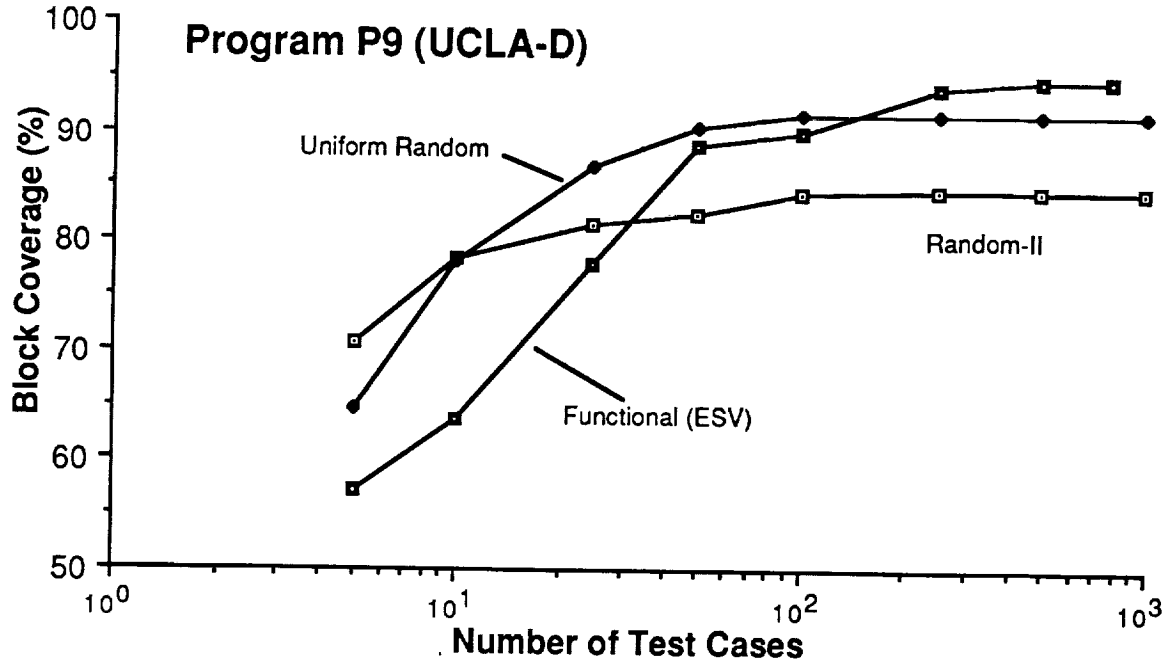


Figure 2.1 Block coverage provided by three "black-box" test data sets.

We see that the lowest cumulative coverage was provided by RANDOM IIdata, while the highest coverage was obtained through functional, or extremal and special value (ESV), test cases designed on the basis of the requirements specification, and the study of the "golden" code. But even these test cases do not provide 100% block coverage. There are two reasons:

a) often programmest have supplied code for unsolicited functions, and
b) code on paths infeasible under the given requirements specifications.

The uniform random profile is a pessimistic one, but tends to provide better coverage because the rarely used functions and code sections have a better chance of being executed. It is interesting to observe that the ESV data provide better coverage than the random data only after a certain number have been generated and executed (over about 200 in this case). The implication is that **if** the generation of the functional test data is an incomplete, low quality, process (e.g. inexperienced test

team/programmers), then it it is quite possible that better code coverage may be achieved by employing uniform random data. Coverage information for all twenty versions is tabulated in Appendix IV.

The fault detection efficiency of the test data we have used is illustrated in Table 2.2. The table shows the number of faults detected by each type of data (ESV and random) during the certification testing. We see that neither ESV nor random data provided perfect detection on their own. They both missed the same number of dissimilar faults. Howevert ESV cases we used detected all similar faults, while random data detected only 5 out of the 7 similar faults. Additional information on the faults can be found in Appendix V, while more information on the fault detection properties of the employed test cases in Appendix VI.

Table 2.2 The number of faults detected by ESV-I and random test data used during phase II certification testing. The ESV-I set contains 796 cases. The random data were composed of 100 cases generated using RANDOM-Ia profile, 100 using RANDOM-Ib profile, and 200 cases generated using RANDOM-II profile (see Appendix II for details).

Test Set	Fault Type		Total
	Dissimilar	Similar	
Random+ESV-I	54	7	61
Random	48	2	50
ESV	48	7	55
Random but not ESV	6	0	6
ESV but not Random	6	5	11

From Table 2.2 we see that random data, if genrated with some care, can be almost as good as special valute test cases in detecting faults. Unfortunately, the efficiency of the random test case can vary. Table A6.1 shows the irder of detection of faults for the ESV-I set. Table A6.2 shows the order of detection of faults for 500 RANDOM-Ib test cases. Table A6.3 summarizes the fault detection propertise of the ESV-I set and the 500 case RANDOM-Ib set. We immediately see that these random cases, on the average, detected only about half of the faults detected by the ESV data. Furthermore from Table A6.2 we see that the fault detection effectiveness of one random data set appers to saturate very quickly. Our experience with the random testing of RSDIMU code is that the sensitivity of the random test cases to errors is low. Unless partitioning is employed (it might be better to use ESV data in that case) detection capabilitiy of the random test cases for distinct

errors saturates extremely quickly. After 2 to 10 random test cases the same errors are usually detected over and over again (if not removed). Once past 100 random test cases the detection of new, different, errors becomes an almost negligible event. This can be changed if we change the random sampling profile is changed and tuned to the character of the already detected faults, or partitions not previously covered are sampled.

It would apppear that a similar effect was observed in the recent experiment reported by [ShL88] where incomplete random back-to-back testing detected about 50% of the faults, while low quality functional testing detected the other half of the faults.

3. Faults

The faults discovered in the investigated code were divided into three classes: specification faults (S), implementation faults (F) and modification faults (M):

* Specification faults are those that can be directly traced to addition of new functions (and therefore lack of these functions in the developed code), and those that can be traced to ambiguities in the original requirements specification document (that resulted in multiple interpretation of parts of the document) and led to changes in the requirements specification document for the purpose of clarifying these ambiguities.
* Implementation faults are non-specification faults that were in the code submitted after the phase I acceptance testing (summer 85), and the faults that were inadvertently introduced by the certification programmers during certification testing (summer 87) as part of the code changes needed to satisfy addition of new functions to the requirements specification document. Implementation faults do not include mistakes made during the correction of specification ambiguity faults.
* Modification faults are those introduced by certification programmers as part of the attempts to correct specification ambiguity faults, implementation faults, and already committed modification faults. Some of the common-cause faults were cross-listed because of their origin and nature. Information on the modification faults is currently available only for the seven programs certified at NCSU.

Table 3.1 lists the prevalence of the faults by their (development) phase of origin. The faults that are cross-listed appear only in the first category into which they fit from the left of the table (e.g. S3/F2 fault of Table 3.2 is counted only in the specification column). The 'Iterations' column indicates the

number of correction cycles certification programmers went through before the final, certified, version of the code was produced. Each iteration cycle ended with a new version of the code. The information for the last two columns of the table was not available for the thirteen programs not certified at NCSU. A total of 3 distinct specification ambiguity related faults, and 58 distinct implementation faults were detected in the twenty investigated programs. In addition, 8 modification faults were detected in the seven programs certified at NCSU. A change in the specification requirements regarding a call to a voter function (not an integral part of the inertial navigation problem) required changes in all 20 versions at the beginning of the certification phase. For completeness this change (S4) is in Table 3.1 but is not considered a true fault. Rounded to the nearest integer, the average number of specification faults per program is 3, while the average number of implementation faults per program is 5. In the seven programs for which the information on modification faults was available we find an average of 1 modification fault per program. At the NCSU site the correction of the faults required about 9 iterations on the average.

Table 3.1 Classification of faults detected in the programs (S4 is not counted).

Actual	Program	Specification	Implementation	Modification	Iterations
ncsua	P1	2	2	-	-
ncsub	P2	3	3	-	-
ncsuc	P3	3	5	1	8
ncsud	P4	3	8	3	12
ncsue	P5	3	5	1	5
uclaa	P6	2	3	-	-
uclab	P7	0	4	-	-
uclac	P8	1	6	-	-
uclad	P9	2	8	1	10
uclae	P10	2	2	-	-
uiuca	P11	0	4	-	-
uiucb	P12	≥ 1[1]	≥ 4[1]	-	-
uiucc	P13	2	7	0	10
uiucd	P14	2	4	-	-
uiuce	P15	2	4	-	-
uvaa	P16	1	5	1	10
uvab	P17	2	4	-	-
uvac	P18	2	5	2	7
uvad	P19	1	5	-	-
uvae	P20	1	4	-	-

[1] There were four major overhauls of the P12 code because the original was very defective (note the 'Percent Change" column in Table 2.1). In the process, at least four implementation faults have been removed.

The faults of particular interest in this study are those that can result in failures that result in an

identical-and-wrong (IAW) response from two or more of the components. These faults are often called similar, or common-cause, faults. Similar faults appear in several versions, affect the same functions and problem variables, similar parts of the code, and may cause the versions to fail with either an IAW response, or with different responses. On the other hand, dissimilar faults are unique to the program where they occur, and most often they manifest as response different from that given by any other version (for further discussion of similar faults see, for example, [Avi84], [Bis86,88], [Kel88]). All dissimilar faults were either implementation of modification faults. However, all specification faults, and only very few of the implementation faults, were of the similar kind. A description of the similar faults is given in Table 3.2. A summary of all faults by programs, and a description of the dissimilar faults is given in Appenix V. The distribution of similar faults across the components is given in Table 3.3. Also given is their span for IAW answer failures, and an estimate of the excitation probability of these failures.

Table 3.2 Similar faults.

Fault code	**Fault description**
S1	Misalignment correction problem (specification misinterpretation/ambiguity)
S2	LINOFFSET and other values not computed for failed sensor on input (specification ambiguity)
S3/F2	Not counting all three edge failures before declaring system or sensor failure (specification ambiguity problem, sensitivity fault in the implementation).
F1	Display round off error.
F4	Division by zero on all failed input sensors (fatal run-time failure in all cases).
F13/M9	Votelinout procedure call placement fault (implementation fault precipitated by S4)
F14	Input from sensor **not** masked to 12 bits (modulo 4096 missing).
S4	missing Votelinout voter call (specification **change/addition**)
M5	Wrong computation of variable SIGMA (threshold for sensor failure comparison)

All three genuine specification faults are ambiguity faults. One of the implementation faults (F2) is so

closely related to a specification ambiguity (S3) that we have cross-referenced it with S3. Four of the implementation faults (excluding F2) result in IAW failures. One common-cause implementation fault (F13) is a direct result of the specification change S4, and since this change was implemented during program certification (maintenance) it is also cross-listed as modification fault M9. All specification faults produced correlated failures. All similar implementation faults that we have detected span two or more versions.

Table 3.3 Distribution of common-cause faults.

Fault	Component	Fault Span	IAW span	Probability
S1	P2, P3, P4, P5, P10, P18, P20	7	4 (P2, P3, P4, P10)	$<10^{-5}$
S2	P1,P2,P3,P4,P5,P6,P9 P10,P13,P14,P15,P17,P18,P19	14	13-14 (less P9, all 14)	<0.02
S3/F2	P1, P2, P3, P4, P5, P6,P8, P9,P12,P13,P14,P15,P16,P17	14	2-10 (See Table 3.4)	≤0.24
F1	P2, P3, P4, P7,P8,P9,P10, P13,P15,P17,P18,P19.	12	4 (P2,P9, P13,17)	$<10^{-4}$
F4	P3, P4, P5,P7, P13,P15,P20.	7	7 (Fatal run-time failures)	$<10^{-5}$
F13/M9	P2, P4, P5, P6,P8 P9, P10, P14, P18	9	4 (P5, P8, P10, P18)	N/A
F14	P1, P2,P3,P5,P8,P9, P11, P16, P17, P19.	10	10 (All fail similarly).	N/A
M5	P3, P17	2	2	<0.24

A very interesting fault is the M5 fault. It is the only clearly common-cause modification fault that we have observed. It results in IAW responses from two of the seven versions in which modification faults were investigated. It also appears as an implementation fault in program P11. Of course, coincident failures of two or more components due to different faults were observed. However, accidental IAW's (different faults but identical response for a test case) were not observed. It must be

stressed though, that we have been monitoring a considerable number of variables (59) at any one time, and therefore the effective cardinality of the output space was quite large thus reducing the the probability of such IAW's [McA87].

As mentioned earlier, the faults which are particularly dangerous are those that produce coincident failures which result in IAW's whose span is such that the voter is confused or mislead into choosing an incorrect answer as the correct one. In Table 3.3 N/A in the 'Probability' column means that the fault was not evaluated in a random setting. From the table we see that for non-specification faults IAW span does not exceed 10 versions. One of the similar implementation faults (F4) induces fatal run-time failures in all seven affected components. Such a fault is not difficult to detect if an appropriate test case is constructed during the testing period because it is self-reporting. However, its effect is drastic if it is left in the code of even one of the versions. In this case, construction of an appropriate test case was very easy and natural. Only two other instances of fatal run-time failures were observed for similar faults (component P13 for S3/F2, and component P3 for F1), although a number of fatal-run time failures were recorded for dissimilar faults.

More interesting is the fact that four out of the seven (NCSU) certification programmers did not put the voter routine (S4 change) in the proper place on the first try. This resulted in the F13/M9 fault. This fault has IAW span of 3, and although this is not critical for the operation of the code in the absence of staged-voting[1], it reflects on the potential reliability of inter-stage voting. Similarly, a specification ambiguity fault that is only important if staged voting is used is S2. Fourteen of the 20 versions had this fault. What is interesting, though, is that its IAW span was observed to vary between 13 and 14 depending on the execution environment initialization procedure. The cause was the initialization mode. If all the program variables were initially set to zero (which is often done by interpreters, and compilers) then fourteen programs exhibited a IAW failure which consisted of returning the initialization value as the acceleration estimates due to erroneous declaration of the system failure. However, when specially selected "garbage" values were used to initialize the variables, only thirteen programs returned this value, while the fourteenth one returned zero because it overrode system initialization through its own, unrequested, initialization code.

Two important faults clearly stemming from specification ambiguities, and resulting in specification clarification changes, are S1 and S3/F2. Seven out of twenty components were involved in the case of S1, and fourteen in the case of S3/F2. The S3/F2 fault could be classified either as an implementation fault (specifications misunderstood or misinterpreted by programmers, faulty

[1]Staged voting is an approach where the code of each component is divided up into several stages. After each stage, voting takes place and the answer that is selected as the correct one is passed on to the next stage of all the components.

problem analysis and implementation, or similar), or as a specification ambiguity since a clarification of the relevant specification text was eventually undertaken. The IAW span of S1 is four, but that of S3/F2 is variable. The maximum observed IAW span for S3/F2 is 14 which is also the maximum IAW span observed for specification faults as a class. However, in practice the failure span of S3/F2 varied between 2 and 10 indicating that the correlation of the failures over the affected components is not 100% [c.f. Vou88].

Figure 3.1 illustrates the variable behavior of the IAW response span of S3/F2 fault. In the example we have used 200 uniform random test cases. The variability is caused by different sensitivity of different versions to the channel noise (sensor failure threshold). In the 200 cases there was not a single event where all 4-tuples formed from the 20 versions (a total of 4845) failed with a IAW, then despite the fact that the fault S3/F2 spanned 14 versions, and occasionally could trigger IAW failures in as many as 10 versions, it could be automatically detected using back-to-back testing with at most four versions. One version (P4) exhibited extra high failure rate. Besides failing concidentally with other versions, on 80 occasions the S3/F2 fault triggered an additional single-version failure. All 14 versions never failed on the same test case though, i.e. the maximum failure span was 13. Thus excluding the 80 P4 failures, there were 18 occurrences of the S3/F2 induced coincident failures in 200 cases.

3.1 Fault Avoidance

The properties of the fault set of special interest are those that can give us information about the methods that would be effective in avoiding them in the final versions of the code. Of the 61 distinct faults discovered during the certification testing, only seven were similar faults. Hence, the majority of faults (54, or 88%) could have been easily detected using back-to-back testing. Back-to-back testing appears to be a cost-effective approach for fault detection in systems where the fraction of similar faults is not in excess of about 15 to 20% of the total number of faults [Vou88]. Of course, back-to-back testing should not be limited to random input data. A certain number of well thought-out ESV (functional) test cases has to be constructed, in order to provide the basic functional and structural coverage of the versions. Unfortunately, any deficiency in the construction of these test cases, or in the application of the testing strategy, can drastically reduce the effectiveness of the testing. Therefore,it is very important to enforce adequate application of the testing strategy, and the use of adequate test data. One possible monitoring mechanism are various coverage metrics. It is our experience that neither full statement (or linear block) coverage, nor full branch coverage, is a sufficient indicator of the test data set adequacy (although far better than none). More sophisticated measures, such as advanced data-flow metrics [e.g. Rap85], should be employed. This can be

supplemented with manual, or automatic, verification of the functional coverage provided by the data (compared to requirements specification).

Table 3.4 Analysis of the failure types resulting from the S3/F2 similar fault over 200 uniform random cases. Total failure span for this fault is 14.

Type	Failure Frequency	Failure Total Span	Partial IAW span	Versions Involved in Partial IAW Failure	Comment*
1	7	13	10	(P1, P2, P5, P6, P9, P12, P14, P15, P16, P17)	
			2	(P4,P8)	
			1	**P13**	**FETF**
2	5	13	8	(P1, P2, P4, P5, P8, P14, P16, P17)	
			2	(P3,P12)	
			2	(P6,P15)	
			1	**P13**	**FETF**
3	2	13	8	(P1, P2, P4, P5, P8, P14, P16, P17)	
			3	(P3,P12,P15)	
			1	(P6)	
			1	**P13**	**FETF**
4	1	10	5	(P2,P5,P8,P14,P16)	
			4	(P1, P6, P15,P17)	
			1	**P13**	**FETF**
5	1	6	2	(P8, P17),	
			2	(P10, P14)	
			1	(P4)	
6	1	2	1	(P4)	
			1	(P10)	
7	80	1	1	(P4)	

* FETF - Fatal Execution Time Failure (causes termination of program execution)

The need for formalization of every step of the development process cannot be overstressed. It has been shown in [Avi88] that the use of formal development techniques can greatly reduce the

incidence of correlated faults. Several of the similar faults detected in this experiment could have been avoided if the unit testing had been more formal, and subject to more stringent control, and/or had the appropriate (and easily derived) test cases been used by individual development teams. For example, the F14 fault, which spanned 10 versions, could be detected either by a trivial inspection of the code, or by an equally trivial test case. The problem was that half of the teams simply did not match the explicitly stated functional requirements with their designs and/or code.

An equally glaring example is provided by the implementation fault F1. The fault affects a non-critical function clearly described in the specification document (rounding of displayed digits, which is very easy to check for correctness). Unfortunately, the fault manifests only for a special combination of input parameters which is not simple to generate (reverse program) for the program as a whole. Hence, despite the fact that the required function was very explicitly stated in the requirements, the team developing the functional (ESV) test set decided not to "bother" with this test case, and because the required function was so "simple" and "obvious" a separate test-bed for testing the display module was not used at first. Because the development process of the "golden" code was not formal enough, the fault was not detected prior to its release. Instead, it was discovered only after explicit inspection (code reading) of the "golden" program during the certification process. A set of five special test cases was then generated explicitly for detection of this fault. The excitation probability of the fault turned out to be between about 1 in 10,000 to 1 in 500,000 for the random profiles we have employed for testing, so an equally large number of test cases had to be run and examined (automatically, of course) to find the special cases. Testing of the rest of the versions then showed that a similarly embarrassing, and disastrous, train of reasoning must have been followed by 12 other teams. Again, because a formal mechanism for checking on the functional completeness of the code, and the test sets, was not used the fault "slipped" through. A partly saving grace is the fact that although F14 spans 12 versions, its IAW span is only 4, so back-to-back testing of any 5-tuple would have detected it if an appropriate test case had been generated.

The third example that strongly points to the need for very formalized development and testing at the unit level is the implementation fault F4. In seven programs this fault induced fatal execution-time failures. It is the result of an attempt to divide by zero in the case when all sensors are assumed to have failed on input. A straightforward, textbook, example of an extremal test case, yet seven teams did not think of checking their code for proper functioning under these circumstances. All teams were asked to submit formal development documents, including a formal test plan and a test log. However, because the experimenters did not have a formal (and possibly automated) way of matching the test cases developed by the teams with the requirements, as well as enforcing full branch and functional coverage at the unit testing level, the fault was left in the code to be discovered

in the system testing phase.

Two of the correlated faults (S2, F13/M9) were related to the placement and use of function calls that for inter-stage voting. Because the concept was not clarified in sufficient detail in the initial specification document, majority of the programming teams misunderstood the need for complete computation of some of the variable values in the situation where a sensor was failed on input. This resulted in the S2 fault. Had the internal states been monitored (using the inter-stage voting, or using a debugger) this omission would have been easy to spot. Alternatively, if the specification had been written using a formal notation (a simple event table, or a decision table, would have sufficed), this ambiguity would not have occurred at all. An associated fault is the F13/M9. It is easy to recognize, either by inspection, or by testing. The fault was introduced into the code as the result of a modification in the specifications (S4). We believe that its primary cause is rooted in the nature of the second testing phase (certification, summer 87). At that time the programmers were aware of the existence of the "golden" code, and of the fact that their code would be subjected to a more stringent system test. This may have reduced their individual testing efforts to the correction of the defects reported by the system test, instead of encouraging individualized (unit level) testing. Furthermore, addition of the voter routine was the first task these programmers were given, and because all the programmers were new to the project they were probably insufficiently familiar with the code, and more prone to making an error.

The remaining two similar faults, S1 and S3/F2, are more of a problem. Both stem from ambiguities in the specifications. Both can, of course, be detected if appropriate test cases are designed and the results analyzed, or if detailed code reading and matching with the requirements is undertaken. And again, formal specification notation may have helped to reduce, or even, avoid these misunderstandings. In addition, direct back-to-back testing would have discovered both faults, but at least a 5-tuple would have to have been used to guarantee detection of both faults by random testing (S2 has IAW span of 4, S3/F2 has variable span between 2 and 10). The question is did these specification ambiguities (recognized in hindsight) somehow draw attention to themselves in during the design process, and could that have been used to avoid these faults in the absence of more sophisticated testing tools and a formal specification language?

4. Summary

We have described an experiment in multiversion testing where (we believe) all faults were identified and classified according to their origin during the development phases of the versions. In twenty

functionally equivalent software versions we have detected a total of 3 specification related faults and 58 implementation faults. Altogether seven of the 61 faults exhibited potential for producing correlated failures. Similar faults were found to span as many as 14 versions, but most of them were trivial and could have been easily detected had appropriate testing and verification strategies been used during unit development and testing. Three of the similar faults were specification ambiguity faults, while four were implementation faults. Only two of the similar faults (both specification faults) were classified as "difficult" to detect and evaluate. Two faults (again both specification related) showed variable IAW response span.

Some fault avoidance strategies that we found would have eliminated most, or even all, of the faults are back-to-back testing, correlation of the programmer queries with specifications, more formal unit development and testing, and measurement of the adequacy and quality of the system test data. It was determined that back-to-back testing of 2-tuples could have been used to eliminate about 90% of the faults. Four of the seven similar faults could have been eliminated by using back-to-back testing of 5-tuples. The rest of the similar faults could have been avoided through more formal design and testing process. In fact, most, (perhaps all) similar faults we have observed, could have been avoided had the specifications been written using more formal notation, the unit testing phase was subject to more stringent standards and control, and better tools for measuring the quality and adequacy of the test data (e.g. coverage) were used.

References

[Avi77] A. Avizienis and L. Chen, "On the Implementation of N-version Programming for Software Fault-Tolerance During Program Execution", Proc. COMPSAC 77, 149-155, 1977.

[Avi84] Avizienis and J.P. Kelly, "Fault-Tolerance by Design Diversity: Concepts and Experiments", Computer, Vol. 17, pp. 67-80, 1984.

[Avi88] A. Avizienis, M.R. Lyu, and W. Schutz, "In Search of Effective Diversity: A six-Language Study of Fault-Tolerant Flight Control Software,", Proc. FTCS 18, pp 15-22, June 1988.

[Bis86] P.G. Bishop, D.G. Esp, M. Barnes, P Humphreys, G. Dahl, and J. Lahti, "PODS--A Project on Diverse Software", IEEE Trans. Soft. Eng., Vol. SE-12(9), 929-940, 1986.

[Bis88] P.G. Bishop, and F.D. Pullen, "PODS Revisited--A Study of Software Failure Behaviour", Proc. FTCS 18, pp 2-8, June 1988.

[Eck85] D.E. Eckhardt, Jr. and L.D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors", IEEE Trans. Soft. Eng., Vol. SE-11(12), 1511-1517, 1985.

[Kel86] J.P.J Kelly, A. Avizienis, B.T. Ulery, B.J. Swain, R.T. Lyu, A.T. Tai, and K.S. Tso, "Multiversion Software Development," Proc. IFAC Workshop SAFECOMP'86, Sarlat, France: pp. 43-49, Oct 1986.

[Kel88] J. Kelly, D. Eckhardt, A. Caglayan, J. Knight, D. McAllister, M. Vouk, "A Large Scale Second Generation Experiment in Multi-Version Software: Description and Early Results",

Proc. FTCS 18, pp 9-14, June 1988.

[Kni86] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the assumption of Independence in Multiversion Programming", IEEE Trans. Soft. Eng., Vol. SE-12(1), 96-109, 1986.

[McA87] D.F. McAllister, C.E. Sun, and M.A. Vouk, "Reliability of Voting in Fault-Tolerant Software Systems for Small Output Spaces", North Carolina State University, Department of Computer Science, Technical Report, TR-87-16, submitted for publication, 1987.

[Pan81] D.J. Panzl, " A Method for Evaluating Software Development Techniques", The Journal of Systems Software, Vol. 2, 133-137, 1981.

[Ran75] B. Randell, "System structure for software fault-tolerance", IEEE Trans. Soft. Eng., Vol. SE-1, 220-232, 1975.

[Rap85] S. Rapps and E.J. Weyuker, "Selecting software test data using data flow information", IEEE Trans. Soft. Eng., Vol. SE-11, 367-375, 1985

[Sco84] R.K. Scott, J.W. Gault, D.F. McAllister and J. Wiggs, "Investigating Version Dependence in Fault-Tolerant Software", AGARD 361, pp. 21.1-21.10, 1984

[Sco87] R.K. Scott, J.W. Gault and D.F. McAllister, "Fault-Tolerant Software Reliability Modeling", IEEE Trans. Software Eng., Vol SE-13, 582-592, 1987.

[ShL88] T.J. Shimeall and N.G. Leveson, "An Empirical Comparison of Software Fault-Tolerance and Fault Elimination," 2nd Workshop on Software Testing, Verification and Analysis, Banff, IEEE Comp. Soc., pp 180-187, July 1988.

[Tso87] K.S. Tso and A. Avizienis, "Community Error Recovery in N-Version Software: A Design Study with Experimentation", Proc. IEEE 17th Fault-Tolerant Computing Symposium, pp 127-133, 1987.

[Vou85] M.A. Vouk, D.F. McAllister, K.C. Tai, "Identification of correlated failures of fault-tolerant software systems", in Proc. COMPSAC 85, 437-444, 1985.

[Vou88] Vouk M.A., "On the Cost of Back-to-Back Testing," Proc.6th Annual Pacific Northwest Software Quality Conference, Lawrence and Craig, Inc., Portland, OR, pp263-282, September 1988

Appendix I

"Static" Reliability Models

Assuming that inputs are selected independently according to some probability distribution function, and faults are static (no corrections take place for the duration of the measurement) we can estimate the operational software reliability, $\hat{R}$, by Nelsons's model

$$\hat{R} = 1 - \frac{n_f}{n}, \tag{A1.1}$$

$$Var(\hat{R}) = \frac{n_f(n-n_f)}{n^3} \tag{A1.2}$$

where n is the number of testing runs, and n_f is the number of failures in n runs. As testing progresses more test cases are executed and the system response is verified. With successful corrections the estimate of the reliability (R) of the system increases. It can be shown [e.g. Dur84, Ehr85, How87] that if N representative random test cases are executed and no failures are found, then an upper bound, p_u, on the failure probability, p = 1-R, of the system at α confidence level is given by the following expression:

$$p_u = 1 - (1-\alpha)^{1/N}. \tag{A1.3}$$

The number, N_t, of test cases (runs) that have to be executed without a failure to show that p is below a limit p_u with confidence level of α is

$$N_t = \frac{\ln(1-\alpha)}{\ln(1-p_u)}. \tag{A1.4}$$

A frequent choice for the confidence level is 95% (α = 0.95). For small p_u, this gives the following approximation

$$N_t \approx \frac{3}{p_u}. \tag{A1.5}$$

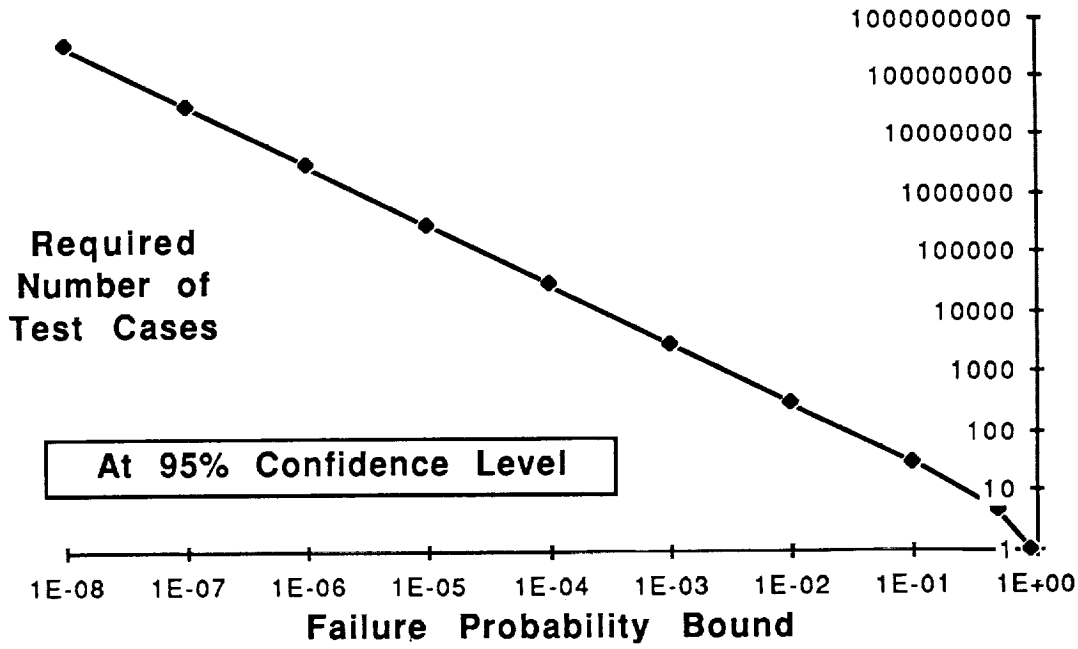


Figure A.1. The number of representative random test cases required to assure with 95% confidence that the true failure probability is less than the bound given on the horizontal axis.

Estimating Static Reliability of RSDIMU Software

The first acceptance testing phase (SUmmer '85) offered 50 random test cases. Assuming that the cases were representative, then the 95% confidence level lower bound on the reliability of a version that has successfully passed all those test cases is 0.94, using the "no failure" model above (equation A1.3).

The second acceptance testing phase (certification testing) included 400 random test cases. Assuming that the cases were representative, then the 95% confidence level lower bound on the reliability of a version that has successfully passed all the random test cases is 0.992, using the "no failure" model above (equation A1.3).

Appendix II

Program Identification and Allocation (appendix not complete)

From fts Tue Mar 3 14:40:17 1987
Received: by cscfac.ncsu.edu (4.12/4.7)
id AA10253; Tue, 3 Mar 87 14:40:11 est
Date: Tue, 3 Mar 87 14:40:11 est
From: fts (fault-tolerant software)
Posted-Date: Tue, 3 Mar 87 14:40:11 est
Received-Date: Tue, 3 Mar 87 14:40:11 est
Message-Id: <8703031940.AA10253@cscfac.ncsu.edu>
To: harvard!crasun!sje@seismo
Subject: ordering of programs
Cc: fts
Status: RO

Tentative ordering of the programs by effort to correct (from NCSU)

N - localized, relatively simple and easy corrections
(no or minimal knowledge of the specs needed)
I - intermediate level errors, probably localized, intermediate
level knowledge of specs may be required
X - extensive, global or difficult corrections. Thorough knowledge
of the specs (expert level) and the program needed

The count in each column gives the number of errors identified for each program on the basis of comparison and classification analysis of acceptance
testing output responses.

Actual amount of code that needs correcting, time actually modifying the code
and the testing visibility of the faults may provide different ordering if used as ordering basis.

It was assumed that knowledge of the specs and the programs is the most timeconsuming factor, at least for new programmers, within the first month of debugging.

	pgm	N	I	X
1	uce	2	0	0
2	ncc	6	0	0
3	uvc	3	1	0
4	uia	4	1	0
5	ncb	2	0	1
6	uve	2	0	1
7	ucb	3	0	1
8	nca	4	0	1
9	uvb	4	0	1
10	nce	5	0	1
11	uie	5	0	1
12	ucd	1	1	1
13	uib	2	1	1
14	ncd	3	1	1
15	uic	3	1	1
16	uva	4	1	1
17	uid	2	1	1
18	uvd	3	1	1
19	uca	3	0	2
20	ucc	3	0	2

Following is a global description of the errors that we believe are present in each program. The list is based on inspection of test responses, and only in very few instances on actual code insepection. We may have missed a few, and we may have claimed an error where there isn't one on its own (it may go away if one of the other fault is cleared). However, in general we believe that the following can serve for ordering programs by difficulty of debugging by programmers new to the project.

The above table was derived from the following.

Fault Classification

X - difficult to correct, I - intermediate, N - localized, easy to correct
FRF - fatal runtime failure

ncsuA9.i
N - missing voter
N - built in g value
N - does not compute linoffset for failed on input (spec. change)
N - display fault (dmode=21)
X - sensitivity problems/differences

ncsuB2.i
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
X - sensitivity problem

ncsuC6.i (FRF)
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
N - calibration failure
N - mod 4096 failure
FRFN - subscript out of range (display)
FRFN - missing label (34)

ncsuD7.i (FRF)
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
FRFN - division by zero (display)
I - display bug
X - sensitivity problem

ncsuE4.i (FRF)
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
FRFN - subscript out of range (display)
N - mod 4096 failure
FRFN - missing label (34)
X - sensitivity problem

uclaA.i (FRF)
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
X - an unknown computational bug, wrong linouts
FRFN - subscript out of range (display)
X - sensitivity problem

uclaB.i (FRF)
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
FRFN - division by zero (display)
X - sensitivity problem

uclaC.i (FRF)
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
FRFN - missing label 45
X - sensitivity problem
X - an unknown computational bug

uclaD.i
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
I - dmode=88 display fault
X - sensitivity problem

uclaE.i
N - missing voter
N - does not compute linoffset for failed on input (spec. change)

uiucA1.i (FRF)
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
I - display (dmode31) fault
FRFN - missing label 99
FRFN - missing label 34

uiucB1.i (FRF)
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
FRFN - label 20 missing
X - potential sensitivity problems

uiucC1.i
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
FRFN - division by zero (display)
FRFN - label 20 missing
I - potential sensitivity problems

uiucD1.i
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
x -sensitivity
I - display problems

uiucE1.i (FRF)
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
FRFN - division by zero
N - calibration fault
N - display fault
X - sensitivity

uvaa1.i
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
N - display fault
X - sensitivity problems
I - computational fult

uvab1.i (FRF)
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
FRFN - label 18 missing
N - display problems
N - mod <4096> problem?
X - sensitivity problems

uvac1.i
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
N - calibration error
I - sensitivity?

uvad1.i
N - missing voter
N - does not compute linoffset for failed on input (spec. change)
N - computationla (sign) error
I - display
X - sensitivity

uvae1.i (FRF)

N - missing voter
N - does not compute linoffset for failed on input (spec. change)
FRFN - label 0 missing
X - a basic problem? won't run more than few cases.

******

programs assigned to NCSU by CRA

ncsuc
ncsud
ncsue
uclad
uiucc
uvaa
uvac

RSDIMU Fault Tolerant Software Experiment
Status Report May 8-15,87
University of California at Santa Barbara

1. UCSB was assigned the following 6 program versions to certify:

ucla.C	difficult
ucla.E	easy
uiuc.A	easy
uiuc.B	difficult
uiuc.D	difficult
uva.B	easy

They have been categorized as easy or difficult to modify by CRA as noted above.

2. I have 11 programmers at my disposal. Some of these are trusted to complete
their assignment, others are not - they were assigned accordingly.

3. Initial Assignments:

ucla.C	t (trusted)
ucla.E	2 * u (untrusted) - this means that 2 programmers were assigned since I was unsure that either one would finish.
uiuc.A	t
uiuc.B	t
uiuc.D	t
uva.B	2 * u

4. Since I still had 3 programmers left, I made the following additional assignments:

uva.A	u

Appendix III

Documentation for RSDIMU-ATS Testing Environment

Fault-Tolerant Software Experiment
ACCEPTANCE ENVIRONMENT
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

This is the basic acceptance environment. Most of the shell scripts and programs have either a help which describes its activation parameters (invoke the script without any parameters), or internal documentation. Where needed program source code is provided. Current version of the system is intended for UNIX csh environment under either 4.2/4.3BSD, or Ultrix 1.1/1.2, running on VAX hardware.

Comparison of the values computed by programs with those using golden code is done using relative tolerance. It is possible to switch to absolute tolerances if that is desired. Do not do that for acceptance testing.

Testing tolerances are set to the following values within fts_accept and can be changed by modifying statements within fts_accept (see fts_accept help):

DiffBestEst = 0.00024414
DiffLinOut = 0.00024414
DiffOffset = 0.00024414
tolerance_mode = relative

Please do not use different tolerances for your acceptance testing before getting concurrence from all the other testing sites. Otherwise we shall each end up testing and correcting different things.

There is no tolerance regarding the display values (five digits), but one could allow for a difference in the last displayed digit. To avoid display related warnings and "failures" of the type: gold 4.9999 vs. computed 5.0000, and to therefore test only the display algorithm, we inject (using voteestimates) golden values for bestest and other real-valued variables prior to display computations.

The acceptance harness tests for agreement on eleven output variables (infact 59, if elements of arrays are counted separately, number of elements is given in parentheses). They are:

LINOFFSET, LINNOISE, LINOUT, LINFAILOUT, SYSSTATUS, BESTEST, CHANEST, CHANFACE, DISMODE, DISUPPER and DISLOWER.

Critical variables are:	(3)BESTEST, (8)LINFAILOUT, {(1)SYSSTATUS}
Non-critical:	(1)DISMODE, (3)DISUPPER, (3)DISLOWER, (12)CHANEST, (4)CHANFACE
Intermediate:	{SYSSTATUS}, (8)LINOFFSET, (8)LINNOISE, (8)LINOUT

All variables are checked for each test case.

For more details on the checking of variables and tolerance used see the listings of the fts_harness files, and the April '86 NCSU Working Notes from the Langley meeting

(NASA.FTS/NCSU/WN/1/Apr-86), and UCLA notes from the same meeting.

To avoid accidental correctness problems output variables are "trashed" before each test case is run. The trash values injected in the various output variables of rsdimu are as follows :

LINOFFSET	-9999.0
LINOUT	999999.0
BESTEST.ACCELERATION [1..3]	9999999.0
CHANEST [1..4].ACCELERATION [1..3]	9999999.0;
DISMODE	65534
DISUPPER [1..3]	65534
DISLOWER [1..3]	65534

The values for real variables (first four listed above) cannot occur for the current set of input data, and are highly unlikely otherwise. The display values are supposed to turn on only the G segment for the least significant digit. Boolean output variables, and user defined are initilized by the compiler (to zero).

Primary scripts:

fts_certify	- shell script which activates fts_accept with all.dat test data and produces a correction request report and test cases for the cerification team by running fts_correq. Certain program naming conventions and running options are built-in.
fts_accept	- shell script for constructing, compiling and running harness+rsdimu code.
fts_correq	- correction request generation shell script, generates a report/request suitable for mailing to the maintenance teams.

Utility scripts and programs:

fts_listdata	- script for listing test cases from the test data files.
fts_prnt	- data listing program.
fts_prterr	- program produces test cases suitable for use by fts_driver.p code.
fts_ter1	- block coverage computation script.
fts_lc	- lower-case filter.
fts_uc	- upper-case filter.
fts_nc	- comment-delimiters lex-based filter.

Source code and script parts:

fts_io	- sed control code to flag "integer","real",and rsdimu i/o.
fts_dbxbug	- dbx bug control code.
fts_dbxinit	- dbx initialization code.
fts_harness.declare	- test harness declarations.
fts_harness.rest	- test harness body.
fts_msgtext	- correction request message.
fts_prnt.p	- source code for fts_prnt.
fts_prterr.p	- source code for fts_prterr

Documentation and examples:

ReadMe	- general information about the "accept" directory.
ReadMe_to_certify	- certification procedure using fts_certify

ReadMe_accept	- using fts_accept.
example/	- directory with example outputs from an fts_accept run (ncsuD7.i tested by executing: fts_accept ncsuD7.i ncd7 all -c -x > test.ncd7all& fts_correq ncd7 all > correq.ncd7).
newcode/	- empty directory for testing results (reminder),

Data links:

all.dat	- symbolic link to esv+random acceptance test cases.
esv.dat	- link to extremal and special value (esv) test cases.
randNCSU.dat	- link to independent random test cases (uniform profile).
randCRA.dat	- link to independent random test cases (shaped profile).

All .dat files are in ../data.

It is also assumed that the code to be tested is in ../code.

The fts_lc, fts_uc, fts_nc, and initial fts_io filters were written by RTI.

The filter fts_io (integer/real, and i/o) is rather crude. It will miss 'real' at the begining of a line. It may also cause false warnings regarding use of integer and real types, and of i/o in the rsdimu code. In those cases hand editing and recompilation of <work_name>.p (rsdimu+harness) files may be necessary. If editing, search for two question marks ??. If recompiling use: pc -s -C -g -z options. Re-run fts_accept without the -c option.

Note that -s compiler option yields messages regarding non-standard use of Pascal in the code (primarily the harness code). These messages should be ignored.

Alternatively, delete the first two lines (real/integer), or third and fourth lines (i/o) of the fts_io code.

fts_nc filter for comments may cause problems by making nested comments of type {(* comment *)} transform to {{ comment }} which is illegal. This filter can be excluded from the processing pipe in fts_accept.

Fault-Tolerant Software Experiment
PROCEDURE FOR ACCEPTANCE TESTING
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

```
**************************************************************************
*                                                                        *
* For the purpose of conducting the acceptance testing (certification    *
* of the 20 programs) it is recommended that you use fts_certify.        *
* Unless you intend to use fts_accept directly, rather than through      *
* the fts_certify facility, you do not need to read this file.           *
*                                                                        *
**************************************************************************
```

****

It is assumed that the communication between the experimenters and the maintenance personnel will be via electronic mail. It is further assumed that the experimenter has full access to maintenance personnel files, but the reverse is not true. It is also assumed that the acceptance testing is performed in csh in UNIX 4.2/4.3BSD, or Ultrix 1.1/1.2 environments running on VAX hardware (else see nonVAX_host/ directory).

I.

The testing begins by placing the code which is to be tested into the code/ directory. Enter the accept/ directory and start the initial round of testing by executing the fts_accept shell script. It is recommended that you use the -x option and benefit from the coverage information thus provided. For example:

```
fts_accept ncsuB2.i ncb2 all -c -x > test.ncb2all&
```

NOTE: YOU CANNOT RUN TWO fts_accept JOBS FROM THE SAME DIRECTORY AT THE SAME TIME (IN BACKGROUND). THE SYSTEM WAS NOT DESIGNED FOR THAT AND YOU CAN END UP WITH A MESS. YOU CAN, HOWEVER, KEEP TWO DIRECTORIES, SAY ACCEPT1 AND ACCEPT2 AND RUN AN fts_accept FROM EACH OF THEM WITHOUT INTERFERENCE.

The run should either result in error messages (exit code <= 8) or should complete successfully (exit code 11). When the background job ends check the test.<name>all file carefully.

* No compiler errors or missing voter call problems (exit status > 5).
* Fatal execution time errors exit status = 7.
* Differences detected from expected output exit status =8.

II.

If the test run ends with any status but exit(11), i.e. complete success, produce an error correction request for the maintenance team by running fts_correq script. For example:

```
fts_correq ncb2 all > correq.ncb2
```

Make sure that the number of failures you wish to analyse is set to 1. For this see fts_correq code (run fts_correq without parameters). File errdata.ncb2 will contain input data for failed cases in a form that is suitable for use by the "fts_driver.p" code in certify/. Check the content of correq.ncb2 and mail it to the maintenance team working on the <name> code (in examples: ncsuB2.i and higher versions, i.e. <name>=ncb2, or ncb3 etc).

You may also have situations where you need to send non-standard messages as part of the correction request. For example, people may send you code and reports with incorrect or inappropriate version numbers. In situations like that create and insert the message at the begining of the correq.<name> file, just after the standard initial paragraph.

III.

Now create a subdirectory that will hold the starting, and all subsequent versions for a particular program(mming team). For example:

mkdir newcode/ncsuB

make a sub-subdirectory for the current program version:

mkdir newcode/ncsuB/v2

and move all the files you want to keep into that sub-sub directory, e.g.

mv *ncb2* newcode/ncsuB/v2

You may wish to use diff and compress processors to reduce stored file sizes.

Unless you are interested in doing further correlation analysis and extracting intensity functions and experimental MCF profiles (for the purpose of detecting and eliminating inter-version dependence during the acceptance testing, not assumed a standard procedure in this experiment) you may not wish to keep trace.<name>all, vect.<name>all and binrep.<name>all files. The ter1.<name>all file contains a compressed overview of the executed code blocks (all begining with 0.---I have not been executed and you should find out why). You will not generate the trace, vect and ter1 files, nor keep binrep if you do not use the -x option. You may also wish to dispense with <name> and <name>.p files which are the executable harness+rsdimu and source harness+rsdimu respectively. We would recommend that you save at least the test.<name>all and the error.<name>all files.

Any communication (questions and answers) received prior to corrected program version are also saved into the "active" newcode sub-sub directory as "q1", "a1", "q2", "a2" etc.

IV.

Upon receiving a message with the location of the latest corrected version, and of the correction/change report(s), cd to accept/:

* Create a new subsubdirectory in the appropriate program subdirectory, e.g. ncsuB3.i location and change report have just been received mkdir newcode/ncsuB/v3
* Save the location/change report message into v<number>, e.g. from inside the mail:

  s <#> newcode/ncsuB/v3/correction_report

  where <#> is the number of the mail message on your h-list.

* Then (<path> points to maintenance team location the code):

  cp <path>/ncsuB3.i newcode/ncsuB/v3/ncsuB3.i
  cp <path>/ncsuB3.i ../code/ncsuB3.i

fts_accept ncsuB3.i ncb3 all -c -x > test.ncb3all&

Now repeat the previous steps depending on the results of the test run, i.e. run a fts_correq if necessary, move results of the run into, for example v3 etc. Use appropriate university name and version numbers.

****

Notes:

****

It is expected that the maintenance team makes a single error correction that was requested by the correq.<name> report and sends back to you a mail message giving the location in their directories of the new and corrected code version, the new version number and one (or more if several changes had to be made to correct an error) error correction report(s). Save the received location message and the correction report into the "active" newcode sub-sub directory as "correction_report", e.g.

newcode/ncsuB/v2/correction_report

You procede then to pick-up the new version of the code and copy it into appropriate newcode sub-sub file, and ../code file (you may wish to use pointers/links to save space instead). Check that they send you the code and the report with an appropriate version numbers everytime.

>>

Check the error report they send against the test report you have!!! If there is any indication at all that the difference may be due to an error in the "golden" code (i.e. supplied expected answers)freeze all testing and immediately inform ATS distribution site, i.e. NCSU (see fts86/ReadMe for address, phone etc.)

>>

It is extremely important for the success of the experiment that you keep not only the final, corrected version of each program, but that each intemediate version submitted for acceptance testing is saved and tagged with an appropriate version number and information about the changes/corrections (using the provided change form). It is expected that programmers will correct one error at a time (and should not be given requests for more than one correction at a time), so that we can keep track of the influence particular errors had on the overall system failure probability etc.

The "certify" directory contains code and files that would be sent to each maintenance/certification team. It contains a basic rsdimu driver (to avoid interface problems) and instructions on its use. It also contains a sample input and output, and an electronic error report file. Whenever a change is made in the code it is expected that the programmer will record using this report. The new version of the code and the error and change report copy are both returned to the experimenters.

It is essential that each program be given a version number and associated with it the date of it creation. Every time a program is corrected its version changes and should be recorded in the correction report, as comments in the code itself, and should reflect in the file name for the new code (as kept in the "code" directory, and in the "newcode" directory in accept/).

****

Fault-Tolerant Software Experiment
PROCEDURE FOR CERTIFICATION TESTING
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

It is assumed that the communication between the experimenters and the maintenance personnel will be via electronic mail. It is further assumed that the experimenter has full access to maintenance personnel files, but the reverse is not true. It is also assumed that the acceptance testing is performed in csh in UNIX 4.2/4.3BSD, or Ultrix 1.1/1.2 environments running on VAX hardware (else see nonVAX_host/ directory).

I.

The testing begins by placing the code which is to be tested into the code/ directory. Enter the accept/ directory and start the initial round of testing by executing the fts_certify shell script. For example:

```
fts_certify ncsuD7.i ncd7
or

fts_certify ncsuD7.i ncd7 > certify.ncd7&
```

The latter form should be used if you wish to run in the background.

NOTE: YOU CANNOT RUN TWO fts_certify JOBS FROM THE SAME DIRECTORY AT THE SAME TIME (EVEN IN BACKGROUND). THE SYSTEM WAS NOT DESIGNED FOR THAT AND YOU CAN END UP WITH A MESS. YOU CAN, HOWEVER, KEEP TWO DIRECTORIES, SAY ACCEPT1 AND ACCEPT2 AND RUN AN fts_certify FROM EACH OF THEM WITHOUT INTERFERENCE.

fts_certify calls fts_accept with all.dat testset and -c option. Output is automatically routed into file test.<name>all. This file will then contain information about the acceptance test run, and will be used as the basis for forming a correction request file correq.<name>.

The run should either result in error messages (exit code <= 8) or should complete successfully (exit code 11). When the job ends check the test.<name>all file carefully.

* No compiler errors or missing voter call problems (exit status > 5).
* Fatal execution time errors exit status = 7.
* Differences detected from expected output exit status =8.

II.

If the test run ends with any status but exit(11), i.e. complete success, an error correction request will be produced for the maintenance team. The correction request will be in correq.<name>, and the test cases that caused up to the first 20 failures will be in errdata.<name>.

Check the content of correq.<name> and mail it to the maintenance team working on the <name> code (in examples: ncsuD7.i and higher versions, i.e. <name>=ncd7, or ncd8 etc).

Mail or copy directly into the directory of the maintenace team file errdata.<name>. The format of the data in this file is suitable for direct use by their "driver" program, so that they can do their own testing.

****

WARNING

Your system may have a byte limit on mail messages that can be passed through it (e.g. 100,000 bytes). In that case you may find that your correction report may be too large, and may become truncated by the e-mail system. It is safer to send only short messages and to transfer long files directly into certification team's directory using cp, e.g. 20 failures in ncsuD7.i generate a correq.ncd7 request file of about 4000 lines of code (about 170,000 bytes)

****

You may also have situations where you need to send non-standard messages as part of the correction request. For example, people may send you code and reports with incorrect or inappropriate version numbers. In situations like that create and insert the message at the begining of the correq.<name> file, just after the standard initial paragraph.

III.

The following procedure describes program and data version management without RCS or SCCS. You should read it regardless of the management procedure you will use so that you can decide what to save/store.

Create a subdirectory that will hold the starting, and all subsequent versions for a particular program(mming team). For example:

mkdir newcode/ncsuD

make a sub-subdirectory for the current program version:

mkdir newcode/ncsuD/v7

and move all the files you want to keep into that sub-sub directory, e.g.

mv *ncd7* newcode/ncsuD/v7

Unless you are interested in doing correlation analysis and extracting intensity functions and experimental MCF profiles (for the purpose of detecting and eliminating inter-version dependence during the acceptance testing, not assumed a standard procedure in this experiment) you may not wish to keep trace.<name>all, vect.<name>all and binrep.<name>all files. The ter1.<name>all file contains a compressed overview of the executed code blocks (all begining with 0.---I have not been executed and you should find out why). You will not generate the trace, vect and ter1 files, nor keep binrep if you do not use the -x option. When using fts_certify this is default in or der to reduce program testing time and sve storage.

You may also wish to dispense with <name> and <name>.p files which are the executable harness+rsdimu and source harness+rsdimu respectively. We would recommend that you save at least the test.<name>all and the error.<name>all files.

To save space you can save only the difference in the code between the starting version and the new version (e.g. ncsuD7.i and ncsuD8.i, or ncsuD7.i and ncsuD9.i). For that purpose use the diff processor (read DIFF(1) manual).

You can further reduce storage space by "compressing" all the saved files using compress, or any other code compression processor available on your machine.

Whatever the scheme, make sure that you can rebuild the starting files and versions.

Any communication (questions and answers) received prior to corrected program version are also saved into the "active" newcode sub-sub directory as "q1", "a1", "q2", "a2" etc.

IV.

Upon receiving a message with the location of the latest corrected version, and of the correction/change report(s), cd to accept/:

* Create a new subsubdirectory in the appropriate program subdirectory, e.g. ncsuD8.i location and change report have just been received

```
mkdir newcode/ncsuD/v8
```

* Save the location/change report message into v<number>, e.g from inside the mail:

```
s <#> newcode/ncsuD/v8/correction_report
```

where <#> is the number of the mail message on your h-list.

* Then (<path> points to maintenance team location the code):

```
cp <path>/ncsuD8.i newcode/ncsuD/v8/ncsuD8.i
cp <path>/ncsuD8.i ../code/ncsuD8.i
```

alternative for latter (saves space):

```
cd ../code
ln -s ../accept/newcode/ncsuD/v8/ncsuD8.i  ncsuD8.i

fts_certify ncsuD8.i ncd8 > certify.ncd8&
```

Now repeat the previous steps depending on the results of the test run. Use appropriate university name and version numbers.

****
Notes:
****

It is expected that the maintenance team makes a error corrections for up to the first 20 reported failures that were requested by the correq.<name>. Certification (maintenance) team is supposed to mail back a message giving the location in their directories of the new and corrected code version, the new version number and one (or more if several changes had to be made) error correction report(s). You may if you wish use hardcopy correction reports, but there is a danger that the reports may evenuatly get separated from the code and corrections to which they refer. Furthermore if kept in electronic form it may be easier to analyse them.

Save the received location message and the correction report(s) into the "active" newcode sub-sub directory as "correction_report", e.g.

```
newcode/ncsuD/v8/correction_report
```

You procede then to pick-up the new version of the code and copy it into appropriate newcode sub-sub file, and ../code file (you may wish to use pointers/links to save space instead). Check that teams send you the code and the report with an appropriate version numbers everytime.

>>>
Check the error report tems send against the test report you have!!! If there is any indication at all that the difference may be due to an error in the "golden" code (i.e. supplied expected answers) freeze all testing and immediately inform ATS distribution site, i.e. NCSU (see fts87/ReadMe for address, phone etc.)
>>>

It is extremely important for the success of the experiment that you keep not only the final, corrected version of each program, but that each intemediate version submitted for acceptance testing is saved and tagged with an appropriate version number and information about the changes/corrections (using the provided change form).

The "certify" directory contains code and files that would be sent to each maintenance/certification team. It contains a basic rsdimu driver (to avoid interface problems) and instructions on its use. It also contains a sample input and output, and an electronic error report file. Whenever a change is made in the code it is expected that the programmer will record it using this report. The new version of the code and the error and change report copy(ies) are returned to the experimenters.

It is essential that each program be given a version number and associated with it the date of its creation. Every time a program is corrected its version changes and should be recorded in the correction report, as comments in the code itself, and should reflect in the file name for the new code (as kept in the "code" directory, and in the "newcode" directory in accept/).

****

Fault-Tolerant Software Experiment
Instructions for Certification Teams
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/12-May-87

Welcome to the FTS certification project. This is a NASA sponsored experiment in Fault-Tolerant Software. Your part will be to find and correct any bugs in the software, under controlled conditions. The software means one of the twenty programs that were produced during the first phase of this experiment.

You should be in possession of the following documentation (if you are not please see your experiment supervisor).

RSDIMU specification (version 3.2/10-Feb-87)
RSDIMU specification (version 2.1/19-Sep-85)

You should also have a directory called "certify" sent to you by the FTS experimenters. In this directory you have the code for the driver, instructions for its use, and some test data. All file that are part of this testing package begin with 'fts_' and should not be modifed by you except under special circumstances, and after consultation with your experiment supervisor. You can of course copy them and then modify them at will. This is not advised.

You will be receiving messages about needed corrections via e-mail. You should read them, correct the program to the best of your abilities, and test it. The data on which the program failed will either be attached to the message or will be sent to you separately. You can then use this data to test your code.

The message with correction requests is expected to be of the form

correq.<name>

where <name> is the code for the program you are correcting (university code followed by the team code and program version number). The data for the cases you failed (suitable for use with your system are expected to in a file of the form

errdata.<name>all

Once you are satisfied that you have found the fault that is causing the error(s) you were requested to correct, you should fill out (make a copy) the error_report and send it to your supervisor's e-mail address. Your site may also require you to fill in and submit a hard copy of the report. In the same message you should also tell us what is the current version of your program and in which file one can find it (we shall need copies of your corrected programs so do not change them once you have sent a message that one is ready for pick-up).

The program which you have just finished correcting must be in a file called <unam>XX.vYY, where <unam> is the agreed upon abbreviation for the university at which the code was originally produced (e.g. ncsu, ucla, uiuc, uva), XX stands for the letter and number associated with your program code, and YY is the current version of your code (you begin by incrementing the number in XX by one). For example C6, i.e. ncsuC6.v07 means that you have updated ncsuC6 to version 7). You should also learn to update the version number in the program header in the style in which it is already there. If it is not part of the code you should add a comment header with the version number and date (e.g. ncsuC6.v07/15-Jul-86). A sample header is shown in fts_driver.p code.
If in doubt please ask about details.

Please bear in mind that the original specifications have been changed and that the latest version (the one you have) may require you not only to correct existing code, but also to add to it (for example missing calls to voter routines).

File ReadMe_data contains a description of the test cases that are being used to test your code. You should use this list in conjunction with the correq.<name> report to locate and identify errors.

Please read the documentation you have received for the experiment. Note that you should keep all communications concerning the program you have been given between yourself and the experimenter, and should communicate through electronic mail

****

e-mail address is: fts

****

Please feel free to ask e-mail questions about any part of this experiment.

Note the following rules and guidelines:

1. Do not change protections on any work-related files.
2. Communications are restricted during this experiment as follows. You may not discuss any aspects of your work in this job with other programmers. Any work-related communication between you and the Professor is to be conducted via UNIX mail. We require this so in the event of an error or ambiguity in the specifications, or some other significant event, all students may be sent a copy of the mailed question and its answer. 3. Every day you work you must log onto your UNIX account. In this way you will receive in a timely manner all mail concerning answers to questions, any updates in the specifications etc. You should also fill in a time-sheet once a day.
4. Your Professor will read the UNIX mail once early in the morning and again in the late evening (Monday through Friday). All questions should be directed to him/her.
5. It is your responsibility to read about UNIX tools you are not familiar or comfortable with.
6. Once a week, on Friday, you should submit a weekly progress report, describing the work you did during the week (number and type of errors you have corrected, any problems you have encountered running the driver harness, hardware problems, the total time you have spent working on the project during the week, whether reading or using the computer, if reading you should specify what and which part of the specs or which error prompted you to that action, etc.). Report is to be submitted via e-mail.

Good luck

Fault-Tolerant Software Experiment
DATA
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

All the testcases in this release of the system comply with the specification version 3.2/10-Feb-87. A test case entry consists of an input record, and an output record. The latter contains what is believed to be the correct answer to the input record according to the 3.2 specs, and as generated/given by gold3v2.i.

Your code will be tested with a set of extremal and special value (ESV) test cases (796 of them) followed by 400 random test cases.

The ESV test cases are based on an initial random case which was then modified step by step to check a particular function and/or option. In the following overview of the ESV data only the principal features of the data sub-classes are given, along with the principal variables that were changed at any one time. Changes are given with respect to a base test case (usually #1).

1 A general random test case, DMODE=0 (RTI, foof1.dat), votecontrol=0. Checks that all the voters are off.

2-7. Test cases checking voter placement. Voters in the fts_harness are activated via VOTECONTROL one at the time. VOTECONTROL activation order 1, 3, 2, 4, 8 16. Votecontrol and their activation results are detailed below:

1 : Sets the Linnoise values of first 4 sensors to true.
3 : Sets Linoffset value of first sensor to 0.0
2 : Sets linout value of first sensor to BADDAT;
4 : Sets SYSSTATUS to false.
8 : sets the estimate values of acceleration values to large values of 99999.0
16 : Sets garbage values in display output values.

8 General test case, (base 1) DMODE = 88.

9-19 base = case #1, systematic changes in DMODE testing for principal display modes (0,21-24,31-33,1,2,99). Check for various display modes which do valid displays, and the boundary display modes.

20-27 Check for mod 4096 (all chans), base = #1, offraw:selected values are increased with 4096 or 8192 to check if only lower order bits are being used.

28-52. base = #1, changes in DMODE and LINFAILIN, checking for different "blank" displays, specific failure display formats, and failures of one sensor (28-37), whole faces (two sensors, 38-43), and various combinations of four failing sensors (44-52), with one instance of eight failed sensors (50). The sensors are failed on input, to check for I display.

53-85. base = #1, random activation of different display modes continues (to ultimately test all values 0-99 by the end of the ESV set). Noise on calibration (OFFRAW) in steps of +/- 6, +/-12, +/- 18 and +/- 24.
Case 57 test has LINSTD=8, DMODE=1 and noise on calibration channel 1 of +/-24 (8x3=24). Also checked are the display failure formats for LINNOISE values, and the correct use of variable LINSTD, and correct computation of calibration

noise levels. 6 and 24 were chosen because these were the boundary cases for noisiness for the linstd values chosen.

87-110 changes in LINSTD (9, 2,1 with +/- 24 on i/p channels) to check correct use of LINSTD variable and sensitivity of the calibration procedure.

86, 111-149 changes in RAWLIN, DMODE, LINFAILIN, various combinations of failures on input, noise and edgevector failures, base = #1. Values in RAWLIN are so changed as to reflect an assured failure in edgevector test, so that there are no ambiguities left. The values of DMODE are again chosen to test the display failure format. The failures are combined with failures on input, to see if the edgevector tests are properly employed.

150-151 Large changes in misalign [i,6] field, only the sixth axis was chosen for contamination because according to the latest specifications that is the only angle not used in the rsdimu procedure. It use significantly changes output values only if its value is much larger than normal. Changes in the values of other angles will not provided new information.

152-392 Test cases checking for the minimal sensor noise levels for failure declaration.
Cases 152-365 no prior failures.
Cases 366-392 prior failures on one and two faces. These test cases test the sensitivity requirements that all three edges fail the edgevector test before a failure is declared. False alarms are raised when only one or two edges fail. The normal value for the triplet threshold is 49 counts away from the correct figure for no prior failures on the rsdimu. The threshold values will change with the number and place of previous failures.

393-796 CRA proposed test cases with various combinations of sensors failed on input and up to one additional sensor failed in the edge vector test. 56 test cases with 1 sensor failed on input. 168 cases with two sensors failed on input. 120 cases with 3 sensors failed on input. 30 cases with 4 sensors failed on input. 8 cases with 7 sensors failed on input. and the rest are other combinations.

Test cases numbers higher than 796 refer to random test cases.

Fault-Tolerant Software Experiment
DRIVER FOR THE RSDIMU CERTIFICATION
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

**** You do not need to read this if you will be using fts_compile and fts_execute macros. ****

fts_driver.p is a Pascal driver program to run your rsdimu procedure.You may make modifications to suit your tastes, but it is adequate in its present form. To compile it you have to include your file containing rsdimu procedure in the place provided in the source code. (It has to have a .i suffix to run successfully). Also you have to make a call to rsdimu procedure at the place designated.

The compiler command would be:

pc -C -s -o driver fts_driver.p

-s option would circumvent any problems you may encounter due to mixed letter cases and non-standard i/o handling.

The executable module is created in file "driver", which can be run as a shell command. The driver expects the testcase input in a format as shown in the file "fts_errdata.sample". The output, after a successful run of the driver, is in fts_sample.out. Note driver is interactive. If you wish to generate your own input data you will need to use the "No_output_data" option.

Note that there are several parameters which are special and are not part of the rsdimu variable/parameter set and are not given in the specs. These variables appear at the beginning of the fts_errdata.sample file. The rsdimu parameters begin with 15.0000 for obase. If you wish to use the fts_driver.p on its own and without golden data then you need to retain only the line before 15.0000 (votecontrol, case number). Votecontrol serves to control special voter routine actions (whether a particular voter changes the values of it parameters or not). It is used solely for testing placement and use of the voter routines. You need to leave it as is for regression testing of your code after correction. You may experiment with it if you wish to build you own test sets. You do not have to worry about it in the rsdimu code, the variable is taken care of in the driver code.

The other parameters control the comparisons with golden answer and you do not need to use them, unless you provide full format of the file (with dummy golden answers for example).

Fault-Tolerant Software Experiment
Testing RSDIMU code
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

Using the fts_compile and fts_execute macros is simple. Run them without any parameters to obtain the description of the paramters you need. The following sample should help.

It is assumed that you have received correq.ncd6 and errdata.ncd6all from your experiment supervisor.

To compile program ncsuD7 which is (let's assume so) in your certify file, and is the program you have just corrected run

fts_compile ncsuD7 7 > c.7&

When the run finishes check c.7 for compilation errors etc. If ok proceed (rsdimu.7 will contain driver+ncsuD7 executable code).

fts_execute 7 errdata.ncd6all ncsuD7 > x.7&

When job finishes check x.7. If there still are differences from the expected outputs go back and correct your code once more, otherwise submit error_reports and the new code to your supervisor. Make use of the correq.ncd6 and ReadMe_data.

Fault-Tolerant Software Experiment
DATA FOR THE ACCEPTANCE TESTING
RSDIMU ACCEPTANCE TESTING SYSTEM (RSDIMU-ATS)
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

All the testcases in this release of the system comply with the specification version 3.2/10-Feb-87. A test case entry consists of an input record, and an output record. The latter contains what is believed to be the correct answer to the input record according to the 3.2 specs, and as generated/given by gold3v2.i.

The test cases are supplied in Pascal readable files. The format can be found in the fts_prnt.p source code in accept/. The format in which the test cases are given is suitable for use with the accept/fts_accept. Use of the system in non-Vax and UNIX-like environments is described in nonVAX_host directory, data may be re-generated using code suplied in the "generators" directory. The latter action is should not be undertaken without consultation with the ATS distribution site (NCSU).

If you wish to print out all, or some, of the test cases use accept/fts_listdata. If you wish to compare (difference) test cases use fts_diff.

This set of test cases was designed and generated for acceptance testing of the rsdimu code. It consists of a group of 796 extremal/special value (ESV) test cases and a group of 400 random test cases. There are four files of data:

all.dat	ESV test cases, followed by random test cases (randomNCSU.dat, then randomCRA.dat).
esv.dat	ESV test cases, only (796).
randNCSU.dat	random test cases, only (uniform sampling, 200).
randCRA.dat	random test cases, only (shaped sampling, 200).

A successful pass through all the test cases gives an estimated lower limit on the reliability of the rsdimu code of about 0.992 (valid for the employed sampling profile).

The all.dat set should provide 100% block coverage of the rsdimu code. If this is not the case (running fts_accept with -x option will give the coverage info), one should very carefully examine the tested code in places where coverage was not provided. The nature of the rsdimu problem, and the specifications, is such that a thorough programmer can provide for situations and functions which are not explicitly handled in the specifications (e.g. singular matrices, large changes in the slope constants leading to large raw acceleration values). Redundant code of the type that cannot be excited according to the current specifications, but could possibly be needed under exceptional circumstances, should be tested by the programmers providing it. They should also provide test cases for these situations (if possible). Alternatively they should provide a written explanation of the cirumstances and reasons for including that particular code. The golden program gold3v1.i, for example has 5 blocks handling display of extremal input acceleration values (>10g) which are not tested by the current acceptance data set since such large input values are outside the conversion range of the provided equations.

The coverage figures should be considered only in the last stage of the acceptance testing, i.e. when all.dat cases have been passed without a failure, and all the corrections requests have been implemented (e.g. after the final regression pass through all.dat).

The ESV data set is further described in the ReadMe_esv file, and the random data sets are described in the ReadMe_random file.

Fault-Tolerant Software Experiment
THE ESV DATA FOR THE ACCEPTANCE TESTING
RSDIMU ACCEPTANCE TESTING SYSTEM (RSDIMU-ATS)
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

ESV-I Set

The data file esv.dat contains 796 extremal and special (ESV) test cases. The test cases were designed to provide full functional coverage of the RSDIMU specifications v3.2/10-Feb-87.

The test cases are based on an initial random case which was then modified step by step to check a particular function and/or option. In the following overview of the ESV data only the principal features of the data sub-classes are given, along with the principal variables that were changed at any one time. Changes are given with respect to a base test case (usually #1). Listing of all or some of the test cases can be obtained by running accept/fts_listdata. Difference between test cases may be examined using fts_diff.

1. A general random test case, DMODE=0 (RTI, foof1.dat), votecontrol=0. Checks that all the voters are off.

2-7. Test cases checking voter placement. Voters in the fts_harness are activated via VOTECONTROL one at the time. VOTECONTROL activation order 1, 3, 2, 4, 8 16.
Votecontrol and their activation results are detailed below:
1 : Sets the Linnoise values of first 4 sensors to true.
3 : Sets Linoffset value of first sensor to 0.0
2 : Sets linout value of first sensor to BADDAT;
4 : Sets SYSSTATUS to false.
8 : sets the estimate values of acceleration values to large values of 99999.0
16 : Sets garbage values in display output values.

8 General test case, (base 1) DMODE = 88.

9-19 base = case #1, systematic changes in DMODE testing for principal display modes (0,21-24,31-33,1,2,99). Check for various display modes which do valid displays, and the boundary display modes.

20-27 Check for mod 4096 (all chans), base = #1, offraw: selected values are increased with 4096 or 8192 to check if only lower order bits are being used.

28-52. base = #1, changes in DMODE and LINFAILIN, checking for different "blank" displays, specific failure display formats, and failures of one sensor (28-37), whole faces (two sensors, 38-43), and various combinations of four failing sensors (44-52), with one instance of eight failed sensors (50). The sensors are failed on input, to check for I display.

53-85 base = #1, random activation of different display modes continues (to ultimately test all values 0-99 by the end of the ESV set). Noise on calibration (OFFRAW) in steps of +/- 6, +/-12, +/- 18 and +/- 24.
Case 57 test has LINSTD=8, DMODE=1 and noise on calibration channel 1 of +/- 24 (8x3=24). Also checked are the display failure formats for LINNOISE values, and the correct use of variable LINSTD, and correct computation of calibration noise levels. 6 and 24 were chosen because these were the boundary cases for

noisiness for the linstd values chosen.

87-110 changes in LINSTD (9, 2,1 with +/- 24 on i/p channels) to check correct use of LINSTD variable and sensitivity of the calibration procedure.

86, 111-149. changes in RAWLIN, DMODE, LINFAILIN, various combinations of failures on input, noise and edgevector failures, base = #1. Values in RAWLIN are so changed as to reflect an assured failure in edgevector test, so that there are no ambiguities left. The values of DMODE are again chosen to test the display failure format. The failures are combined with failures on input, to see if the edgevector tests are properly employed.

150-151 Large changes in misalign [i,6] field, only the sixth axis was chosen for contamination because according to the latest specifications that is the only angle not used in the rsdimu procedure. It use significantly changes output values only if its value is much larger than normal. Changes in the values of other angles will not provied new information.

152-392. Test cases checking for the minimal sensor noise levels for failure declaration.
Cases 152-365 no prior failures.
Cases 366-392 prior failures on one and two faces.These test cases test the sensitivity requirements that all three edges fail the edgevector testi before a failure is declared. False alarms are raised when only one or two edges fail. The normal value for the triplet threshold is 49 counts away from the correct figure for no prior failures on the rsdimu. The threshold values will change with the number and place of previous failures.

393-796 CRA proposed test cases with various combinations of sensors failed on input and up to one additional sensor failed in the edge vector test. 56 test cases with 1 sensor failed on input. 168 cases with two sensors failed on input. 120 cases with 3 sensors failed on input. 30 cases with 4 sensors failed on input. 8 cases with 7 sensors failed on input. and the rest are other combinations.

More detailed information about the ESV test cases can be obtained by displaying the differences between a chosen base case (#1 usually) and a series of other test cases. Utility shell script fts_diff, based on the UNIX diff processor, is provided for this purpose. By executing fts_diff esv.dat esvdiff 115 123 1

you can obtain, for example, in file esvdiff differences in the input values of cases 115 to 123 with respect to test case #1 of the data file esv.dat.

The CRA document regarding choice of random and ESV test cases was provided as a separate item (not in electronic form) with release 2.0 of RSDIMU-ATS.

Fault-Tolerant Software Experiment
THE RANDOM DATA FOR THE ACCEPTANCE TESTING
RSDIMU ACCEPTANCE TESTING SYSTEM (RSDIMU-ATS)
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

This set of 400 random test cases for rsdimu code is provided primarily for the purpose of estimating the lower limit on the reliability of the tested code, and as a check on the completeness of the ESV test cases. The test cases are completely independent, and no attempt was made to mimic a flight trajectory and the associated time correlation among the input variable values. Therefore any cumulative effects linked to time correlation or auto- correlation remain untested, here and in the extremal and special value (esv.dat) set.

The random data are provide in two sub-sets: randomNCSU.dat and randomCRA.dat.

****

randomNCSU.dat (RANDOM-I profile)

Within the employed sampling domain the distribution of the generated input values is essentially flat. It contains 200 test cases.

In all cases the random data were generated using the random number generator provided with the Pascal comiler (pc, UNIX/Ultrix). Details of the mapping from the random numbers into the actual input variables are given below. More details will be supplied on request. The part of the code used to generate random input values is also enclosed (as fts_NCSUzzrand.i), and it derives in part from the RTI random test harness (September 85).

The input variables randomly sampled, or computed on the basis of randomly sampled values are: offraw, linfailin, rawlin, misalign, normface, temp, phiv, thetav, psiv, phi, thetai, psii, dmode, linnoise, linfailout, scale0,1,2, obase, linstd and nsigt.

A more thorugh understanding of the random generation process and of the resulting input profiles can be gained by studying the fts_NCSUzzrand.i code.

The random set, randomNCSU.dat, consists of two hundred random test cases stratified into two sub-sets. The first one hundred test cases (**RANDOM-Ia**) have the noise on sensors (rawlin) boosted by 200 counts everytime linfailout for a sensor is true. Thus the sensor noise level is guaranteed to exceed the sensitivity threshold of about 50 counts and the sensor should be recognized as failed. The second one hundred test cases (**RANDOM-Ib**) have the noise added as a uniform distribution between 1 and maxnoise-1 counts, and at half the uniform frequency for 0 and maxnoise, the latter value having been read in by the driver program. In this particular case maxnoise was 110, therefore the added noise was symmetrically centered around the threshold value of 55 counts.

It is important to note that random test cases are intended to run after all ESV test cases have been successfully negotiated. There are special situations and combinations of variable values that are covered in ESV test cases and not covered by the sampling domain used to generate present random test cases. Our experience with the random testing of rsdimu code is that the sensitivity of the random test cases to errors is very low. Unless very detailed partitioning is employed (better to use ESV cases in that case) detection capabilities of the random test cases to distinct errors saturate extremely quickly. After 2 to 10 random test cases the same errors are usually detected over and over again (if not removed). Once past 100 random test cases detection of new, different, errors becomes an almost negligible event, unless the random sampling profile is changed and tuned to the character of the already detected faults, or partitions not previously covered are sampled.

For all practical purposes the two sets of 100 test cases, are a single random set of 200 test cases, which if executed successfully, provides us with a lower limit for the rsdimu reliability (at the 95% confidence level) of about 0.985.

initial random seed for 1st 100 cases is: 777
initial random seed for 2nd 100 cases is: 1234567890

****

randomCRA.dat (RANDOM-II profile)

The second sub-set, randCRA.dat, was generated on the basis of the CRA document TM8602/26-Aug-86.

The CRA random test cases are generated with the specifications provided in the CRA documents (especially for PHIV, PSIV and THETAV, NSIGT = 2..7 etc). The calibration noise is normally distributed, and the number of noisy sensors during calibration is exponentially distributed with a parameter of 0.18. The edge vector test can fail one additional sensor, with random noise of upto 200 counts. No sensors fail on input. Generation details can be found in fts_CRAzzrand.i

initial random seed is: 987654321

****

For all practical purposes the two sets of 200 test cases, are a single random set of 400 test cases, which if executed successfully, provides us with a lower limit for the rsdimu reliability (at the 95% confidence level) of about 0.992.

During the generation of the random test cases care is taken to examine the obtained data and to eliminate cases where more than one sensor fails in flight.

Fault-Tolerant Software Experiment
THE GOLDEN DISPLAY AND RSDIMU CODES
RSDIMU-ATS 3.0/PR/UNIX/FTS.NASA-LaRC.Va/NCSU.CSC/05-Mar-87

There are 2 files in this directory viz. display.i, and gold3v1.i The sources correspond to specification version v3.2/10-Feb-87,

display.i contains the display module extracted from the gold program. It has been extensively tested as a stand alone module, and gives results in accordance with the specifications v3.2. It does not need any special declarations in the main program except for those which are in the RSDIMU procedure assumed to be globally available (only the type declarations.) To use this procedure just use standard #include compiler option, and the calling format

display (DISMODE, DISUPPER, DISLOWER);

Use of the golden rsdimu follows the same rules as use of any other rsdimu code, and is fully explained in the specs (see also certify/driver.p).

Appendix IV

Code Coverage

Tables A4.1-A4.6 represent the coverage information for all the twenty final versions of the RSDIMU software. Highest coverage is obtained from the ESV-I test cases, since they were designed to cover all the aspects of computations involved. Because sets of only 1000 random cases were used, random coverage fiugres are lower. The RSDIMU-I profile provides better coverage than the more realistic RSDIMU-II data generation profile.

Program (Total blocks)	Number of cases							
	5	10	25	50	100	250	500	1000
NCSUA (513)	107/4	84/1	68/1	67/1	63/1	62/1	62/1	62/1
NCSUB (315)	82/3	59/2	49/1	45/1	44/1	44/1	44/1	44/1
NCSUC (544)	225/4	204/2	157/1	141/1	110/1	93/1	93/1	93/1
NCSUD (343)	60/2	47/1	34/0	34	34	34	34	34
NCSUE (475)	141/3	122/1	92/1	91/1	87/1	86/1	86/1	86/1
UCLAA (338)	82/3	67/1	58/1	54/1	51/1	50/1	50/1	50/1
UCLAB (368)	106/5	88/1	66/1	58/1	56/1	56/1	56/1	56/1
UCLAC (378)	90/6	62/1	46/1	43/1	42/1	42/1	42/1	42/1
UCLAD (432)	127/4	103/1	80/1	76/1	67/1	66/1	66/1	66/1
UCLAE (288)	75/3	60/3	44/2	44/2	44/2	44/2	44/2	44/2
UIUCA (263)	62/4	45/2	34/2	32/2	32/2	32/2	32/2	32/2
UIUCB (260)	54/4	38/0	33/0	32	32	32	32	32
UIUCC (261)	79/5	67/3	51/3	50/3	48/3	46/3	46/3	46/3
UIUCD (313)	91/2	82/1	64/1	62/1	59/1	59/1	59/1	59/1
UIUCE (351)	65/1	60/1	47/1	43/1	41/1	41/1	41/1	41/1
UVAA (469)	107/3	74/1	51/1	46/1	44/1	43/1	43/1	43/1
UVAB (436)	126/1	103/7	58/1	53/1	46/1	44/1	44/1	44/1
UVAC (918)	654/28	637/26	575/22	563/21	515/18	469/16	468/1	468/16
UVAD (319)	82/5	68/2	49/2	45/2	40/2	38/2	38/2	38/2
UVAE (501)	197/5	191/3	108/0	101	100	97	97	97

Table A4.1 Block/Procedures Non-Coverage Information for 1000 RANDOM-II cases. Values shown are the blocks not executed during testing. Following '/' is the number of unexecuted procedures. If there is no value for the Procedures count, it indicates that all procedures were executed.

Program	Number of cases							
	5	10	25	50	100	250	500	1000
NCSUA	79.14	83.62	86.74	86.93	87.71	87.91	87.91	87.91
NCSUB	73.96	81.26	84.44	85.71	86.03	86.03	86.03	86.03
NCSUC	58.63	62.50	71.13	74.08	79.77	82.90	82.90	82.90
NCSUD	82.50	86.29	90.08	90.08	90.08	90.08	90.08	90.08
NCSUE	70.31	74.31	80.63	80.84	81.68	81.89	81.89	81.89
UCLAA	75.73	80.17	82.84	84.02	84.91	85.20	85.20	85.20
UCLAB	71.19	76.08	82.06	84.23	84.78	84.78	84.78	84.78
UCLAC	76.19	82.53	87.83	88.62	88.88	88.88	88.88	88.88
UCLAD	70.60	78.47	81.48	82.40	84.49	84.72	84.72	84.72
UCLAE	73.95	79.16	84.72	84.72	84.72	84.72	84.72	84.72
UIUCA	76.42	82.88	87.07	87.83	87.83	87.83	87.83	87.83
UIUCB	79.23	85.38	87.30	87.69	87.69	87.69	87.69	87.69
UIUCC	69.73	74.32	80.45	80.84	81.60	82.37	82.37	82.37
UIUCD	70.92	73.80	79.55	80.19	81.15	81.15	81.15	81.15
UIUCE	81.48	82.90	86.60	87.74	88.31	88.31	88.31	88.31
UVAA	77.18	84.22	89.12	90.19	90.61	90.83	90.83	90.83
UVAB	71.10	76.37	86.69	87.84	89.44	89.90	89.90	89.90
UVAC	28.75	30.61	37.36	38.67	43.89	48.91	49.01	49.01
UVAD	74.29	78.68	84.63	85.89	87.46	88.08	88.08	88.08
UVAE	60.67	61.87	78.44	79.84	80.03	80.63	80.63	80.63

Table A4.2 Block Coverage Information for 1000 RANDOM-II cases (percentage of blocks covered).

Program (Total blocks)	Number of cases							
	5	10	25	50	100	250	500	796
NCSUA (513)	168/13	148/9	87/0	32	28	22	22	22
NCSUB (315)	120/6	96/2	47/0	17	11	7	7	7
NCSUC (544)	249/5	221/3	165/1	102/0	96	83	83	83
NCSUD (343)	87/4	71/3	43/1	13/0	7	3	3	3
NCSUE (475)	163/3	148/3	106/1	72/1	67/1	54/1	54/1	54/1
UCLAA (338)	120/7	96/3	54/0	26	22	17	17	17
UCLAB (368)	152/10	118/4	69/0	23	16	14	14	14
UCLAC (378)	149/11	105/6	51/0	28	20	11	10	10
UCLAD (432)	185/12	157/7	96/0	49	44	26	23	23
UCLAE (288)	96/4	78/2	52/1	15/0	12	6	3	2
UIUCA (263)	78/7	63/5	34/1	17/1	14/1	8/1	8/1	8/1
UIUCB (260)	75/8	59/4	28/0	24/0	20	17	17	17
UIUCC (261)	93/7	84/5	55/2	26/2	23/2	16/2	16/2	16/2
UIUCD (313)	142/10	113/5	83/1	47/1	43/1	28/1	28/1	28/1
UIUCE (351)	79/3	64/2	49/1	27/1	26/1	16/1	10/0	10
UVAA (469)	158/5	114/3	73/1	34/1	28/1	11/1	11/1	11/1
UVAB (436)	161/14	141/11	89/6	40/6	36/6	11/0	11	11
UVAC (918)	717/31	699/29	648/27	422/11	394/10	314/6	181/0	51
UVAD (319)	102/7	91/5	65/2	42/2	41/2	26/2	15/2	14/2
UVAE (501)	252/10	238/8	218/6	80/3	75/3	48/0	20	12

Table A4.3 Block/Procedures Non-Coverage Information for 796 ESV-I cases (Actual values are the blocks/procedures not executed during testing).

Program	Number of cases							
	5	10	25	50	100	250	500	796
NCSUA	67.25	71.15	83.04	93.76	94.54	95.71	95.71	95.71
NCSUB	61.90	69.52	85.07	94.60	96.50	97.77	97.77	97.77
NCSUC	54.22	59,37	69.66	81.25	82.35	84.74	84.74	84.74
NCSUD	74.63	79.30	87.46	96.20	97.95	99.12	99.12	99.12
NCSUE	65.68	68.84	77.68	84.84	85.89	89.05	89.05	89.05
UCLAA	64.49	71.59	84.02	92.30	93.49	94.97	94.97	94.97
UCLAB	58.69	67.93	81.25	93.75	95.65	96.19	96.19	96.19
UCLAC	60.58	72.22	86.50	92.59	94.70	97.08	97.08	97.08
UCLAD	57.17	63.65	77.77	88.65	89.81	93.98	94.67	94.67
UCLAE	66.66	72.91	81.94	94.79	95.83	97.91	98.95	99.30
UIUCA	70.34	76.04	87.07	93.53	94.67	96.95	96.95	96.95
UIUCB	71.15	77.30	89.23	90.76	92.30	93.46	93.46	93.46
UIUCC	64.36	67.81	78.92	90.03	91.18	93.86	93.86	93.86
UIUCD	54.63	63.89	73.48	84.98	86.26	91.05	91.05	91.05
UIUCE	77.49	81.76	86.03	92.30	92.59	95.44	97.15	97.15
UVAA	66.31	75.69	84.43	92.75	94.02	97.65	97.65	97.65
UVAB	63.07	67.66	79.58	90.82	91.74	97.47	97.47	97.47
UVAC	21.89	23.85	29.41	54.03	57.08	65.79	80.28	94.44
UVAD	68.02	71.47	79.62	86.83	87.14	91.84	95.29	95.61
UVAE	49.70	52.49	56.48	86.02	87.02	90.41	96.00	97.60

Table A4.4 Block Coverage Information for 796 ESV-I cases (percentage of blocks covered).

Program (Total blocks)	Number of cases							
	5	10	25	50	100	250	500	1000
NCSUA (513)	151/9	83/0	53	30	29	29	29	29
NCSUB (315)	93/2	56	36	20	19	19	19	19
NCSUC (544)	231/4	171/1	130/1	78	63	62	62	62
NCSUD (343)	73/2	32	21	6	6	6	6	6
NCSUE (475)	161/5	122/3	82/1	66/1	61/1	61/1	61/1	61/1
UCLAA (338)	95/4	66/2	42	24	22	22	22	22
UCLAB (368)	109/4	67	36	25	24	24	24	24
UCLAC (378)	101/6	46	26	16	14	14	14	14
UCLAD (432)	153/7	94	57	42	36	36	36	36
UCLAE (288)	86/3	50/1	23	13	9	9	9	9
UIUCA (263)	66/5	38/1	20/1	13/1	13/1	13/1	13/1	13/1
UIUCB (260)	57/4	31	21	19	19	19	19	19
UIUCC (261)	86/5	57/2	34/2	21/2	20/2	20/2	20/2	20/2
UIUCD (313)	109/5	78/1	46/1	32/1	32/1	32/1	32/1	32/1
UIUCE (351)	76/3	54/2	26/1	18/1	18/1	18/1	18/1	18/1
UVAA (469)	125/3	80/1	33/1	18/1	15/1	15/1	15/1	15/1
UVAB (436)	150/17	85/6	34	20	18	18	18	18
UVAC (918)	676/27	607/22	452/15	317/7	252/5	252/5	252/5	252/5
UVAD (319)	97/6	64/3	44/2	36/2	33/2	33/2	33/2	33/2
UVAE (501)	248/8	179/3	87	56	38	38	38	38

Table A4.5Block/Procedures Non-Coverage Information for 1000 RANDOM-I cases (Actual values are the blocks not executed during testing).

Program	Number of cases							
	5	10	25	50	100	250	500	1000
NCSUA	70.56	87.71	93.56	94.15	94.34	94.34	94.34	94.34
NCSUB	70.47	82.22	88.57	93.65	93.96	93.96	93.96	93.96
NCSUC	57.53	68.56	76.10	85.66	88.41	88.60	88.60	88.60
NCSUD	78.71	90.67	93.87	98.25	98.25	98.25	98.25	98.25
NCSUE	66.10	74.31	82.73	86.10	87.15	87.15	87.15	87.15
UCLAA	71.89	80.47	87.57	92.89	93.49	93.49	93.49	93.49
UCLAB	70.38	81.79	90.21	93.47	94.02	94.02	94.02	94.02
UCLAC	73.28	87.83	93.12	95.76	96.29	96.29	96.29	96.29
UCLAD	64.58	78.24	86.80	90.27	91.66	91.66	91.66	91.66
UCLAE	70.13	82.63	92.01	95.48	96.87	96.87	96.87	96.87
UIUCA	74.90	85.55	92.39	95.05	95.05	95.05	95.05	95.05
UIUCB	78.07	88.07	91.92	92.69	92.69	92.69	92.69	92.69
UIUCC	67.04	78.16	86.97	91.95	92.33	92.33	92.33	92.33
UIUCD	65.17	75.07	85.30	89.77	89.77	89.77	89.77	89.77
UIUCE	78.34	84.61	92.59	94.87	94.87	94.87	94.87	94.87
UVAA	73.34	82.94	92.96	96.16	96.80	96.80	96.80	96.80
UVAB	65.59	80.50	92.20	95.41	95.87	95.87	95.87	95.87
UVAC	26.36	33.87	50.76	65.46	72.54	72.54	72.54	72.54
UVAD	69.59	79.93	86.20	88.71	89.65	89.65	89.65	89.65
UVAE	50.49	64.27	82.63	88.82	92.41	92.41	92.41	92.41

Table A4.6 Block Coverage Information for 1000 RANDOM-I cases (percentage of blocks covered).

Appendix V

Faults

Table A5.1 Faults by programs

NCSUA : S2, S3/F2, S4, F14, F38.
NCSUB : S1, S2, S3/F2, S4, F1, F13, F14.
NCSUC : S1, S2, S3/F2, S4, F1, F3, F4, F5, F14, M1.
NCSUD : S1, S2, S3/F2, S4, F1, F4, F6, F8, F9, F10, F11, F13, M2, M6, M8.
NCSUE : S1, S2, S3/F2, S4, F4, F12, F12a, F13, F14, M3.
UCLAA : S2, S3/F2, S4, F13, F39, F40.
UCLAB : S4 (MAY be F14), F1, F4, F41/M5, F42.
UCLAC : S3/F2, S4, F1, F13, F14, F43, F44, F44a.
UCLAD : S2, S3/F2, S4, F1, F13, F14, F15, F16, F17, F19, F20, M4.
UCLAE : S1, S2, S4, F1, F13.
UIUCA : S4, F14, F45, F46, F47.
UIUCB : S3/F2, S4, Major changes in code triggering errors on each case.
UIUCC : S2, S3/F2, S4, F1, F4, F22, F23, F24, F26, F27.
UIUCD : S2, S3/F2, S4, F13, F48, F49, F50.
UIUCE : S2, S3/F2, S4, F1, F4, F51, F52.
UVAA : S3/F2, S4, F14, F28, F29, F36, F37, M5.
UVAB : S2, S3/F2, S4, F1, F14, F53, F54.
UVAC : S1, S2, S4, F1, F13, F31, F33, F35, M5, M7.
UVAD : S2, F1, F14, F55, F56, F57.
UVAE : S1, S4, F4, F58, F59, F60.

Table A5.2 Classification of dissimilar (distinct) faults detected in the programs.

Actual	Program	Specification	Implementation	Modification
ncsua	P1	0	1	-
ncsub	P2	0	0	-
ncsuc	P3	0	2	1
ncsud	P4	0	5	3
ncsue	P5	0	2	1
uclaa	P6	0	2	-
uclab	P7	0	2	-
uclac	P8	0	3	-
uclad	P9	0	5	1
uclae	P10	0	0	-
uiuca	P11	0	3	-
uiucb	P12	0	4	-
uiucc	P13	0	5	0
uiucd	P14	0	3	-
uiuce	P15	0	2	-
uvaa	P16	0	4	0
uvab	P17	0	2	-
uvac	P18	0	3	1
uvad	P19	0	3	-
uvae	P20	0	3	-

Table A5.3 Description of Simplex Faults

Fault Code	Fault Description	Probability of Occurrence.
F3	Does not isolate failed sensors properly.	< 0.346
F5	Display of acceleration 31-34. (fatal error).	< 0.029
F6	Numerical accuracy problem (conversion from ft. to m)	NA
F8	Miscalculation of bestest acceleration values.	1.00
F9	Miscalculation of LINFAILOUT. (derivative of F2).	< 0.24
F10	Display of acceleration wrong.	< 0.016
F11	Display of hexadecimal values wrong.	< 0.021
F12	Miscomputation of channel estimates.	< 0.24
F12a	Wrong values in display upper.	< 0.081
F15	Improper isolation and detection of failed sensor (derivative of F2).	< 0.24
F16	Miscomputation of channel estimates, if a sensor fails.	< 0.074
F17	Miscomputation of acceleration values.	< 0.42
F19	Wrong values used for faulty channels.	< 0.321
F20	Display of acceleration wrong.	< 0.017
F22	Fatal run time error when displaying 0.0 acceleration.	< 0.009
F23	Miscomputation of channel estimates.	
F24	Fatal run time error due to isolation of bad face. (derivative of F2).	< 0.24
F26	Fatal run time error during display.	< 0.018
F27	Wrong values passed through disupper.	< 0.342
F28	Miscomputation of acceleration display values.	< 0.031
F29	same as F28, but at other place.	< 0.006
F31	Fault in detecting failure of sensors.	< 0.489
F33	Display values miscomputed in dsupper [3].	< 0.043
F35	same as F33, at different place.	< 0.012
F36	Does not subtract gravity vector while computing LINOFFSET.	1.0
F37	Subtracts gravity vector from estimated force.	1.0
F38	Uses predefined value for constant G (32.0)	1.0
F39	Miscomputation of channel estimates.	< 0.562
F40	Miscomputation of acceleration values.	< 0.559
F41	miscomputation of SIGMA values (also see M5).	< 0.389
F42	Miscomputation of acceleration values.	< 0.513
F43	miscomputation of channel estimates, if a sensor fails.	< 0.024
F44	Miscomputation of LINFAILOUT values.	< 0.089
F44a	Miscomputation of LINOFFSET values.	< 0.991
F45	Display errors. (may be more than one).	NA
F46	Miscomputation of channel estimates.	< 0.554
F47	Miscomputation of LINOFFSET values.	< 0.876
F48	Errors in display routines.	NA
F49	Miscomputation of acceleration values.	< 0.774

continued

Table A5.3 (continued) Description of dissimular faults.

Fault Code	Fault Description	Probability of Occurrence.
F50	miscomputation of LINOFFSET.	< 0.818
F51	Display error.	NA
F52	miscomputation of estimates.	< 0.659
F53	display error.	NA
F54	miscomputation of LINOFFSET.	< 0.899
F55	miscomputation of channel estimates.	< 0.587
F56	miscomputation of acceleration values.	< 0.619
F57	wrong values used for G.	1.0
F58	Display error.	NA
F59	miscomputation of channel estimates.	< 0.665
F60	Miscomputation of SIGMA.	< 0.389
F61- F64	Major overhauls in uiucb code in key procedures	1.0

Table A5.4 Description of Modification Faults

Fault Code	Description
M1	Error in display
M2	Display round off error
M3	SYSSTATUS faulty when previous sensor failure
M4	Display of acceleration faulty
M5	Computation of Sigma
M7	Miscomputation of Sigma
M6	Does not lead to any additional error
M8	Conversion of LINSTD to engineering units

Appendix VI

Fault Detection

Table A6.1 The order of detection of faults by ESV-I cases.

PROGRAM	FAULT	CASE#	FAILURES
NCSUA	F38	#1	
NCSUB	F13	#4	
	S2	#10	
	F14	#20	
	S1	#150	
	S3/F2	#160	
NCSUC	S2	#10	
	F14	#20	
	F5	#42	
	F4	#50	
	F3	#55	
	S1	#150	
	S3	#382	
NCSUD	F6	NOT TRIGGERED	
	F13	#4	
	S2	#10	
	F11	#28	
	F10	#45	
	F4	#50	
	F8/M10	#112	
	S1	#150	
	F9	#160	
	S3	#160	
NCSUE	F13	#4	
	S2	#10	
	F14	#20	
	F12a	#46	
	F12	#46	
	F4	#50	
	S1	#150	
	S3	#155	

continued

Table A6.1 (continued-1) The order of detection of faults by ESV-I cases.

PROGRAM	FAULT	CASE#	FAILURES
UCLAA	F13	#4	
	S2	#10	
	F39	#46	
	F40	#112	
	S3	#160	
UCLAB	F13	#4	
	F14	#20	
	F43	#46	
	S3/F2	#155	
	F44	#160	
	F44a	#160	
UCLAD	F20	#1	
	F19	#2	
	F13	#4	
	S2	#10	
	F14	#20	
	F17	#30	
	F16	#44	
	F15	#114	
	S3/F2	#160	
UCLAE	F13	#4	
	S2	#10	
	S1	#150	
UIUCA	F14	#20	
	F45	#28	
	F46	#46	
	F47	#160	
UIUCB	F61	#1	
UIUCC	F22,23	NOT TRIGGERED	
	S2	#10	
	F26	#15	
	F27	#30	
	F4	#50	
	S3	#155	
	F24	#155	

continued

Table A6.1 (continued-2) The order of detection of faults by ESV-I cases.

PROGRAM	FAULT	CASE#	FAILURES
UIUCD	F13	#4	
	S2	#10	
	F48	#21	
	F49	#112	
	S3	#160	
	F50	#160	
UIUCE	S2	#10	
	F51	#28	
	F52	#46	
	F4	#50	
	S3	#155	
UVaA	F36	#1	
UVaB	S2	#10	
	f14	#20	
	f53	#28	
	S3/F2	#155	
	F54	#160	
UVaC	F31	NOT TRIGGERED	
	F13	#4	
	S2	#10	
	F35	#12	
	F33	#15	
	S1	#150	
UVaD	F57	#1	
UVaE	F60	NOT TRIGGERED	
	F58	#28	
	F59	#46	
	F4	#50	
	S1	#150	

Table A6.2 The order of detection of faults by 500 cases of type RANDOM-Ib.

PROGRAM	FAULT	CASE#	FAILURES
NCSUA	F38		
NCSUB	S3	#5	
	F2	#5	
NCSUC	F5	#2	
	F3	#3	
	S3/F2	#69	
NCSUD	F10	#5	
	S3/F2	#5	
	F9	#5	
	F11	#7	
NCSUE	F129	#2	
UCLAA	F40	#1	
	F39	#3	
	S3/F2	#5	
UCLAB	F42	#1	
UCLAC	F44a	#1	
	F43	#5	
	S3/F2	#5	
	F44	#7	
UCLAD	F/16,20	#3	
	F19, 17	#5	
	S3/F2	#5	
UCLAE	S2	#1	

continued

Table A6.2 (continued) The order of detection of faults by 500 cases of type RANDOM-Ib.

PROGRAM	FAULT	CASE#	FAILURES
UIUCA	F45	#2	
	F47	#2	
	F46	#3	
UIUCB	F61	#1	
UIUCC	S3/F2	#5	
	F23	#5	
	F27	#7	
UIUCD	S2	#1	
	F48	#3	
	S3/F2	#5	
	F49	#5	
	F50	#5	
UIUCE	F52	#1	
UVaA	F36	#1	
UVaB	S2	#1	
	F53	#3	
	S3/F2	#5	
	F54	#5	
UVaC	F33	#7	
UVaD	F57	#1	
UVaE	F58	#2	
	F59	#3	

Table A6.3 The number of faults detected by ESV and 500 RANDOM-Ib cases.

Actual	Program	ESV-I	500 RANDOM-Ib	Total
ncsua	P1	4	2	4
ncsub	P2	5	1	6
ncsuc	P3	7	3	8
ncsud	P4	9	3	11
ncsue	P5	8	3	8
uclaa	P6	5	3	5
uclab	P7	3	1	4
uclac	P8	6	4	7
uclad	P9	9	35	10
uclae	P10	3	1	4
uiuca	P11	4	3	4
uiucb	P12	4+	1+	4+
uiucc	P13	6	3	9
uiucd	P14	6	5	6
uiuce	P15	5	3+	6
uvaa	P16	6	1	6
uvab	P17	5	4	6
uvac	P18	5	1	7
uvad	P19	1+,(6)?	1	6
uvae	P20	4	2	5